Lotus

LotusScript 3.1 RELEASE

Cross-Product BASIC Scripting Language

LANGUAGE REFERENCE

Contents

Preface

The *LotusScript Language Reference* is a comprehensive summary of the LotusScript® language. It is also available as on-line help in all Lotus® products that support LotusScript.

Besides this language reference, Lotus provides the following documentation of LotusScript:

- The *LotusScript Programmer's Guide*, a general introduction to LotusScript that describes the language's basic building blocks and how to put them together to create applications.
- The documentation that accompanies each of the Lotus products that support LotusScript. This documentation describes the application development environment as well as the various extensions to the language that the individual product provides.

Typographical conventions

The *LotusScript Language Reference* follows certain typographical conventions in its code examples and syntax diagrams. These conventions are summarized in the following two tables.

Code examples

Item	*Convention*	*Example*
Apostrophe (')	Introduces a comment.	' This is a comment.
Underscore (_)	Signifies that the current line of code continues on the following line.	Dim anArray(1 To 3, 1 To 4) _ As String
Colon (:)	Separates discrete statements on the same line	anInt% = anInt% * 2 : Print anInt%
Keyword	Begins with a capital letter. May contain mixed case.	Print UCase$("hello")
Variable	Begins with a lowercase letter. May contain mixed case.	anInt% = 5
Procedure	Begins with a capital letter. May contain mixed case.	Call PrintResults()

Syntax diagrams

Type face or character	*Meaning*	*Example and comment*
Bold	Items in bold must appear exactly as shown.	**End** [*returnCode*] The keyword End is required.
Italics	Items in italics are placeholders for values that you supply.	**End** [*returnCode*] You can specify a return code by entering a value or expression after the keyword End.
Square brackets [ ]	Items enclosed in square brackets are optional.	**End** [*returnCode*] You can include a return code or not, as you prefer.
Vertical bar |	Items separated by vertical bars are alternatives: you can choose one or another.	**Resume** [**0** | **Next** | *label*] You can enter the value 0, the keyword Next, or a label after the keyword Resume. Since these items are enclosed in square brackets, you can also choose to enter none of these.
Braces { }	Items enclosed in braces are alternatives: you have to choose one. Items within braces are always separated by vertical bars.	**Exit** { **Do** | **For** | **ForAll** | **Function** | **Property** | **Sub** } You have to enter one of the following keywords after the keyword Exit: Do, For, ForAll, Function, Property, or Sub.
Ellipsis	Items followed by an ellipsis can be repeated. If a comma precedes an ellipsis (,...), you have to separate repeated items by commas.	**ReDim** *arrayName***(***subscript*,...**)** You can specify multiple subscripts in a **ReDim** statement.

Running examples

Many topics in this reference manual include one or more LotusScript code examples, under a heading such as “Example” or “Example 1”. These programming examples illustrate specific language features of LotusScript.

In general terms, you can turn an example in this book into a LotusScript application by placing the executable code in a procedure. You run the example by executing that procedure. The following instructions describe the main steps in this process when you are running LotusScript in any of several Lotus products, including Lotus Forms®, Lotus Notes®, Word Pro™, and Freelance Graphics®. The details of how you prepare an example to run depend on the programming environment and the content and intent of the particular example.

1. Navigate to the Declarations script in an empty script module.

For example, in a new form in the Forms Designer, select the Script tab, and select Form from the Object drop-down list box and Declarations from the Proc drop-down list box.

In a Notes™ Agents view (or in any Notes view or document), choose Agent from the Create menu, enter a name for the agent, and select the Script option button. The (Declarations) script is selected in the Event drop-down list box.

In a new Word Pro document or Freelance Graphics presentation, select Edit-Script-Show Script Editor, and then select (Declarations) from the Script drop-down list box.

2. Type the example into the script editor.

   If the example includes executable code that is not in a procedure, put the executable code in a sub. For example, type "RunExample Sub" in the line above the first line of executable code, and type "End Sub" after the last line.

   If the example includes one or more procedures followed by a call to a procedure, delete the call statement, but note which procedure is being called. This is the startup procedure.

   In Lotus Forms, type the example into Declarations.

   In Notes, type the example into (Declarations). Notes automatically separates procedures into separate procedure scripts.

   In other Lotus products, do the following:

   - Type module-level declarations into (Declarations).
   - Type Option statements into (Options).
   - For each procedure, use the Script-New Sub or Script-New Function command to create the new procedure, and type the contents of the procedure into the script editor.

   Caution If you are entering a product script that already exists (such as Initialize or Terminate), select the script from the drop-down list box, and then type the script into the script editor.

3. Run the example.

   For example, in Lotus Forms, put a call to the startup procedure in the Form object NewForm script, and choose View-Filler Mode (or press F7).

   In Notes, put a call to the startup procedure in the Initialize script, choose File-Save, and choose the agent name from the Actions menu.

   In Word Pro or Freelance Graphics, select RunExample in the Script drop-down list box, and use the Script menu to run the sub (or press F5). If the procedure includes parameters, put a call to the procedure in the Initialize script, and then use the Script menu (or press F5) to run the Initialize script.

See your product's documentation for details on writing and running LotusScript applications in that programming environment.

The examples in this reference manual also appear in LotusScript Help as secondary topics attached to many Help topics. To run an example starting from Help, you copy and paste the example text appropriately, in a process similar to the one described above for examples in this manual. The process is described in instructions accessed by clicking in the icon bar of each example topic in Help. (You can also find these instructions as a separate topic in Help by searching for “Examples” or “Running examples”.)

Part 1
Scripting Basics

Chapter 1
Script and Statement Construction Rules

This chapter describes the rules for writing the basic elements of a script in the LotusScript language.

Script and statement construction rules

The following rules govern the construction of statements in a script.

- The statements of a script are composed of lines of text. Each text element is a LotusScript keyword, operator, identifier, literal, or special character.
- The script can include blank lines without effect.
- The text on a line can begin at the left margin or be indented without affecting the meaning.
- Within a statement, elements are separated with white space: spaces or tabs. Where white space is legal, extra white space can be used to make a statement more readable, but it has no effect.
- Statements typically appear one to a line. A newline marks the end of a statement, except for a block statement; the beginning of the next line starts a new statement.
- Multiple statements on a line must be separated by a colon (:).
- A statement, except for a block statement, must appear on a single line unless it includes the line-continuation character underscore (_), preceded by white space.
- The line-continuation character must appear at the end of a line to be continued, preceded by at least one space or tab. Only white space or inline comments (those preceded with an apostrophe) can follow the underscore on the line. Line continuation within a literal string or a comment is not permitted.

Examples: Script and statement construction rules

```
' One statement on one line
Print "One line"

' One statement on two lines; extra white space
Print "One" & _    ' Comment allowed here
   "Two"

' Two statements on one line
Print "One" : Print "Two"
```

Literal number construction rules

The following rules govern the construction of literal numbers in a script.

Kind of literal	*Example*	*Description*
Decimal integer	777	The legal range is the range for Long values. If the number falls within the range for Integer values, its data type is Integer; otherwise, its data type is Long.
Decimal	8	The legal range is the range for Double values. The number's data type is Double.
Scientific notation	7.77**E+**02	The legal range is the range for Double values. The number's data type is Double.
Binary integer	**&B**1100101	The legal range is the range for Long values. A binary integer is expressible in 32 binary digits of 0 or 1. Values of &B100000 ... (31 zeroes) and larger represent negative numbers.
Octal integer	**&O**1411	The legal range is the range for Long values. An octal integer is expressible in 11 octal digits of 0 to 7. Values of &O20000000000 and larger represent negative numbers. Values of &O40000000000 and larger are out of range.
Hexadecimal integer	**&H**309	The legal range is the range for Long values. A hexadecimal integer is expressible in eight hexadecimal digits of 0 to 9 and A to F. Values of &H80000000 and larger represent negative numbers.

Literal string construction rules

The following rules govern the construction of literal strings in a script.

- A literal string in LotusScript is a string of characters enclosed in one of the following sets of delimiters:
  - A pair of double quotation marks (“ ”)
  - A pair of vertical bars (| |)
  - Open and close braces ({ })
- A literal string can include any character.
- Strings enclosed in vertical bars or braces can span multiple lines.
- To include one of the closing delimiter characters “, |, or } as text within a string delimited by that character, double it.
- The empty string has no characters at all; it is represented by ””.
- Strings delimited by vertical bars, braces, or double quotation marks cannot be nested.

Examples: Literal string construction rules

```
"A quoted string"
|A bar string|
{A brace string}
|A string
   on two lines|
|A bar string with a double quote " in it|
"A quoted string with {braces} and a bar | in it"
"A quoted string with ""quotes"" in it"
|A bar string with a bar || in it|
{A brace string with {braces}} in it}
```

Identifier construction rules

An identifier is the name you give to a variable, a constant, a type, a class, a function, a sub, or a property.

The following rules govern the construction of identifiers in a script.

- The first character in an identifier must be an uppercase or lowercase letter.
- The remaining characters must be letters, digits, or underscore (_).
- A data type suffix character (%, &, !, #, @, or $) can be appended to an identifier. It is not part of the identifier.
- The maximum length of an identifier is 40 characters, not including the optional suffix character.
- Names are case-insensitive. For example, VerED is the same name as vered.
- Characters with ANSI codes higher than 127 (that is, those outside the ASCII range) are legal in identifiers.

Escape character for illegal identifiers

Lotus product and OLE classes may define properties or methods whose identifiers use characters not legal in LotusScript identifiers. Variables registered by Lotus products might also use such characters. In these cases, prefix the illegal character with a tilde (~) to make the identifier valid.

Examples: Identifier construction rules

```
' $ is illegal as character in identifier
Call ProductClass.LoMethod$             ' Illegal
Call ProductClass.LoMethod~$            ' Legal

X = OLEClass.Hi@Prop                    ' Illegal
X = OLEClass.Hi~@Prop                   ' Legal
```

Labels

A label gives a name to a statement.

A label is built in the same way as an identifier. It is followed by a colon (:). It can't be suffixed with a data type suffix character.

The following statements transfer control to a labeled statement by referring to its label:

- GoSub
- GoTo
- If...GoTo
- On Error
- On...GoSub
- On...GoTo
- Resume

These rules govern the use of labels in a script:

- A label can only appear at the beginning of a line. It labels the first statement on the line.
- A label can appear on a line by itself. This labels the first statement starting after the line.
- A given statement can have more than one label preceding it; but the labels must appear on different lines.
- A given label can't be used to label more than one statement in the same procedure.

Keywords

A keyword is a word with a reserved meaning in the LotusScript language. The keywords name LotusScript statements, built-in functions, built-in constants, and data types. The keywords New and Delete can be used to name subs that you can define in a script. Other keywords are not names, but appear in statements: for example, NoCase or Binary. Some of the LotusScript operators are keywords, such as Eqv and Imp.

You cannot redefine keywords to mean something else in a script, with one exception: they can name variables within a type, and variables and procedures within a class.

The following are all the LotusScript keywords.

Abs	DefDbl	Input	Minute	Sin
Access	DefInt	Input$	MkDir	Single
ACos	DefLng	InputB	Mod	Space
ActivateApp	DefSng	InputB$	Month	Space$
Alias	DefStr	InputBox	Name	Spc
And	DefVar	InputBox$	New	Sqr
Any	Delete	InputBP	Next	Static
Append	Dim	InputBP$	NoCase	Step
As	Dir	InStr	NoPitch	Stop
Asc	Dir$	InStrB	Not	Str
ASin	Do	InStrBP	NOTHING	Str$
Atn	Double	Int	Now	StrCompare
Atn2	Else	Integer	NULL	String
Base	ElseIf	Is	Oct	String$
Beep	End	IsArray	Oct$	Sub
Bin	Environ	IsDate	On	Tab
Bin$	Environ$	IsElement	Open	Tan
Binary	EOF	IsEmpty	Option	Then
Bind	Eqv	IsList	Or	Time
ByVal	Erase	IsNull	Output	Time$
Call	Erl	IsNumeric	PI	TimeNumber
Case	Err	IsObject	Pitch	Timer
CCur	Error	IsScalar	Preserve	TimeValue
CDat	Error$	IsUnknown	Print	To
CDbl	Evaluate	Kill	Private	Today
ChDir	Event	LBound	Property	Trim
ChDrive	Execute	LCase	Public	Trim$
Chr	Exit	LCase$	Put	TRUE
Chr$	Exp	Left	Random	Type
CInt	FALSE	Left$	Randomize	TypeName
Class	FileAttr	LeftB	Read	UBound
CLng	FileCopy	LeftB$	ReDim	UCase
Close	FileDateTime	LeftBP	Rem	UCase$
Command	FileLen	LeftBP$	Remove	UChr
Command$	Fix	Len	Reset	UChr$
Compare	For	LenB	Resume	Uni
Const	ForAll	LenBP	Return	Unicode
Cos	Format	Let	Right	Unlock
CSng	Format$	Lib	Right$	Until
CStr	Fraction	Like	RightB	Use
CurDir	FreeFile	Line	RightB$	UseLSX
CurDir$	From	List	RightBP	UString
CurDrive	Function	ListTag	RightBP$	UString$
CurDrive$	Get	LMBCS	RmDir	Val
Currency	GetFileAttr	Loc	Rnd	Variant
CVar	GoSub	Lock	Round	Weekday
DataType	GoTo	LOF	RSet	Wend

Special characters

LotusScript uses special characters, such as punctuation marks, for several purposes:

- To delimit literal strings
- To designate variables as having particular data types
- To punctuate lists, such as argument lists and subscript lists
- To punctuate statements
- To punctuate lines in a script

Note Special characters within literal strings are treated as ordinary text characters.

The following table summarizes the special characters used in LotusScript:

Character	*Usage*
“” (quotation mark)	Opening and closing delimiter for a literal string on a single line.
| (vertical bar)	Opening and closing delimiter for a multi-line literal string. To include a vertical bar in the string, use double bars (||).
{ } (braces)	Delimits a multi-line literal string. To include an open brace in the string, use a single open brace ({). To include a close brace in the string, use double close braces (}}).
: (colon)	(1) Separates multiple statements on a line. (2) When following an identifier at the beginning of a line, designates the identifier as a label.
$ (dollar sign)	(1) When suffixed to the identifier in a variable declaration or an implicit variable declaration, declares the data type of the variable as String. (2) When prefixed to an identifier, designates the identifier as a product constant.
% (percent sign)	(1) When suffixed to the identifier in a variable declaration or an implicit variable declaration, declares the data type of the variable as Integer. (2) When suffixed to either the identifier or the value being assigned in a constant declaration, declares the constant’s data type as Integer. (3) Designates a compiler directive, such as %Rem or %If.
& (ampersand)	(1) When suffixed to the identifier in a variable declaration or an implicit variable declaration, declares the data type of the variable as Long. (2) When suffixed to either the identifier or the value being assigned in a constant declaration, declares the constant’s data type as Long. (3) Prefixes a binary (&B), octal (&O), or hexadecimal (&H) number. (4) Designates the string concatenation operator in an expression.

continued

Character	*Usage*
! (exclamation point)	(1) When suffixed to the identifier in a variable declaration or an implicit variable declaration, declares the data type of the variable as Single. (2) When suffixed to either the identifier or the value being assigned in a constant declaration, declares the constant's data type as Single.
# (pound sign)	(1) When suffixed to the identifier in a variable declaration or an implicit variable declaration, declares the data type of the variable as Double. (2) When suffixed to either the identifier or the value being assigned in a constant declaration, declares the constant's data type as Double. (3) When prefixed to a literal number or a variable identifier, specifies a file number in certain file I/O statements and functions.
@ (at sign)	(1) When suffixed to the identifier in a variable declaration or an implicit variable declaration, declares the data type of the variable as Currency. (2) When suffixed to either the identifier or the value being assigned in a constant declaration, declares the constant's data type as Currency.
***** (asterisk)	(1) Specifies the string length in a fixed-length string declaration. (2) Designates the multiplication operator in an expression.
( ) (parentheses)	(1) Groups an expression, controlling the order of evaluation of items in the expression. (2) Encloses an argument in a sub or function call that should be passed by value. (3) Encloses the argument list in function and sub definitions, and in calls to functions and subs. (4) Encloses the array bounds in array declarations, and the subscripts in references to array elements. (5) Encloses the list tag in a reference to a list element.
. (period)	(1) When suffixed to a type variable or an object reference variable, references members of the type or object. (2) As a prefix in a product object reference, designates the selected product object. (3) As a prefix in an object reference within a With statement, designates the object referred to by the statement. (4) Designates the decimal point in a floating-point literal value.
.. (two periods)	Within a reference to a procedure in a derived class that overrides a procedure of the same name in a base class, specifies the overridden procedure.
[ ] (brackets)	Delimit names used by certain Lotus products to identify product objects.

continued

Character	*Usage*
, (comma)	(1) Separates arguments in calls to functions and subs, and in function and sub definitions. (2) Separates bounds in array declarations, and subscripts in references to array elements. (3) Separates expressions in Print and Print # statements. (4) Separates elements in many other statements.
; (semicolon)	Separates expressions in Print and Print # statements.
' (apostrophe	Designates the beginning of a comment. The comment continues to the end of the current line.
_ (underscore)	When preceded by at least one space or tab, continues the current line to the next line.

White space is needed primarily to separate names and keywords, or to make the use of a special character unambiguous. It is not needed with most non-alphanumeric operators. It is sometimes incorrect to use white space around a special character, such as a data type suffix character appended to a name.

Scope of declarations

Scope is the context in which a variable, procedure, class, or type is declared. Scope affects the accessibility of an item's value outside the context in which it was declared. For example, variables declared within a procedure are typically not available outside of the scope of that procedure.

LotusScript recognizes three kinds of scope:

- Module scope
- Procedure scope
- Type or class scope

Name conflicts and shadowing

Two variables or procedures with the same name cannot be declared in the same scope. The result is a *name conflict*. The compiler reports an error when it encounters a name conflict in a script.

Variables or procedures declared in different scopes can have the same name. When such a name is used in a reference, LotusScript interprets it as referring to the variable or procedure declared in the innermost scope that is visible where the reference is used.

A variable or procedure of the same name declared at a scope outside of this innermost visible scope, is not accessible. This effect is called shadowing: the outer declaration(s) of the name are *shadowed,* or made invisible, by the inner declaration.

Module scope

A variable is declared in module scope if the declaration is outside of any procedure, class, or type definition in the module. The variable name has a meaning as long as the module is loaded.

The variable name is visible anywhere within the module. Everywhere within the module, it has the meaning specified in the declaration, except within a procedure, type, or class where the same variable name is also declared.

The variable is Private by default and can be referred to only within the module that defines it. A variable can be referred to in other modules only if it is declared as Public and the other modules access the defining module with the Use statement.

The following situations result in a name conflict across modules:

- Two Public constants, variables, procedures, types, or classes with the same name
- A Public type with the same name as a Public class
- A Public module-level variable with the same name as a Public module-level constant or procedure
- A Public module-level constant with the same name as a Public module-level procedure

The following situations result in a name conflict within a module:

- A type with the same name as a class
- A module-level variable with the same name as a module-level constant or procedure
- A module-level constant with the same name as a module-level procedure

Procedure scope

A variable is declared in procedure scope if it is declared within the definition of a function, a sub, or a property. Only inside the procedure does the variable name have the meaning specified in the declaration. The variable name is visible anywhere within the procedure.

Ordinarily, the variable is created and initialized when the procedure is invoked, and deleted when the procedure exits. This behavior can be modified with the Static keyword:

- If the variable is declared with the Static keyword, its value persists between calls to the procedure. The value is valid as long as the module containing the procedure is loaded.

- If the procedure itself is declared Static, the values of all variables in the procedure (whether explicitly or implicitly declared) persist between calls to the procedure.

The following situations result in a name conflict within a procedure:

- Two procedure arguments with the same name
- Two labels with the same name
- Two variables with the same name
- A procedure argument and a variable with the same name
- A function that contains a variable or argument of the function name
- A property that contains a variable of the property name

Type or class scope

A variable is declared in type or class scope if it is declared within the definition of a type or a class (for classes, it must additionally be declared outside the definition of a procedure). The variable is called a member variable of the type or class.

- *Type member variables*: A type member variable is created and initialized when an instance of that type is declared. It is deleted when the type instance or instance variable goes out of scope.

  The visibility of a type member variable is automatically Public.

- *Class member variables*: A class member variable is created and initialized when an instance of that class is created. It is deleted when the object is deleted.

  Each class member variable can be declared Public or Private. A Private member can only be referred to within the class or its derived classes; class member variables are Private by default.

The visibility of a type member variable (which is always Public) and of a Public class member variable depends, for any particular type or object, on the declaration of the instance variable that refers to that instance:

- If the instance variable is declared Private, then the member variable is visible only in the owning module.
- If the instance variable is declared Public, then the member variable is visible wherever the instance variable is visible. It can be referred to in the other modules where the module that owns this instance variable is accessed with the Use statement.

The following situations result in a name conflict within a type:

- Two type members with the same name

The following situation results in a name conflict within a class:

- Two class members (variables or procedures) with the same name

Implicit declaration of variables

A name is implicitly declared if it is used (referred to) when it has not been explicitly declared by a Dim statement, and also has not been declared as a Public name in another module that is used by the module where the name is referred to.

LotusScript declares the name as a scalar variable, establishing its data type by the following rules:

- If the name is suffixed by a data type suffix character, that determines the variable data type
- If no data type suffix character is specified in the first use of the name, the data type is determined by the applicable Def*type*, if any
- If there is no suffix character and no applicable Def*type*, the variable is of type Variant

If a variable is implicitly declared, it must be used exactly as it first appears: either with or without the data type suffix character.

Once a variable has been implicitly declared, you cannot explicitly declare it in the same scope.

An implicit declaration cannot be used to override an existing declaration of the same variable in an outer scope.

Examples: Implicit declaration of variables

Example 1

```
penguint = 3        ' Implicit declaration of penguint, assuming
                    ' penguint has not been previously declared.
                    ' Data type is Variant, assuming
                    ' there is no Deftype statement specifying the
                    ' data type of names beginning with letter p.
```

Example 2

```
penguint% = 3       ' Implicit declaration of penguint.
                    ' Data type suffix character %
                    ' declares data type as Integer.
Print penguint%     ' Prints 3
Print penguint      ' Error. Since penguint was implicitly declared
                    ' with suffix character %, all references must
                         ' specify penguint%, not penguint.
```

Data type conversion

LotusScript implicitly converts data from one type to another in the situations described in the following paragraphs.

You can also convert data types explicitly using the functions CCur, CDat, CDbl, CInt, CLng, CSng, CStr, and CVar.

Numeric operations

When numeric values with different data types are used in a numeric operation, LotusScript converts the values to the same data type for evaluation.

In general, LotusScript converts a data type earlier in this ordering to a data type later in this ordering: Integer, Long, Single, Double, Currency. For example, in an operation with one Integer operand and one Double operand, LotusScript converts the Integer value to a Double before evaluating the expression.

Specific rules for conversion in operations are detailed in the documentation of the individual operators.

Argument passing

When a numeric argument is passed by value to a procedure, LotusScript tries to convert the value if it is not of the data type that the procedure expects. If the value is too large to fit in the expected data type, the operation generates an error.

When a numeric argument is passed by reference to a procedure, the data type of the reference must match that of the declared argument, unless the declared argument is of type Variant.

Variant variables

LotusScript does not routinely convert numbers to strings or vice versa. However, when a value is contained in a Variant variable, LotusScript tries to convert the value to a number or a string, depending on the context.

For example, the Abs built-in function requires a numeric argument. If you pass the string value “-12” in a String variable as an argument to the Abs function, it generates an error. If you pass the string value in a Variant variable, LotusScript converts it to the integer value -12.

Constants

LotusScript provides several built-in constants that you can use in your scripts. These are described in the following table.

LotusScript predefines other constants in the file lsconst.lss. To include this in your scripts, use the %Include directive.

Constant	*Description*
NULL	Represents unknown or missing data. Only Variant variables can take the NULL value. To test a variable for the NULL value, use the IsNull function. You can use the NULL value to represent errors in your script, or the absence of a result. The result of an expression containing a NULL operand is typically NULL. Many built-in functions return NULL when they are passed a NULL value.
NOTHING	The initial value of an object reference variable. As soon as you assign an object to the variable, the variable no longer contains NOTHING. When the object is deleted, the value of the variable returns to NOTHING. You can explicitly assign the value NOTHING to an object reference variable. To test a variable for the NOTHING value, use the Is operator.
True	The Boolean value TRUE, which evaluates to the numeric value -1. This value can be returned by a comparison or logical operation. In an If, Do, or While statement, which tests for TRUE or FALSE, any nonzero value is considered TRUE.
False	The Boolean value FALSE, which evaluates to the numeric value 0. This value can be returned by a comparison or logical operation.
PI	The ratio of the circumference of a circle to its diameter.

LotusScript also includes an internal value named EMPTY. This is the initial value of a Variant variable. If converted to a string, it is the empty string (“”). If converted to a number, it is 0 (zero).

To test a variable for the EMPTY value, use the IsEmpty function.

There is no keyword to represent EMPTY.

Chapter 2
Procedures

Procedures are named sections of a script that you can invoke by name. A procedure in LotusScript takes the form of a function, a sub, or a property.

A Function statement defines a function. The Function statement also declares the function name and the data type of its return value; and the types of the function parameters, if any.

A sub is defined by a Sub statement. The statement also declares the sub name, and the data types of the sub parameters.

A property is defined by either or both of a Property Get statement and a Property Set statement. The defining statement also declares the property name and the data type of the property. A property has no parameters.

A function returns a value; a sub does not. A property has a value.

Any procedure can be forward declared. The forward declaration enables the procedure to be used before the procedure is defined.

The default data type of any function or sub parameter, and of a function's return value, is Variant.

Examples: Defining procedures

```
' Define the function TestFunc and the sub TestSub.

Function TestFunc (X As Integer, Y As Integer) As Integer
   TestFunc = X * Y
End Function

Sub TestSub (X As Integer, Y As Integer)
   Print X * Y
End Sub

' Call the function TestFunc and the sub TestSub.
Print TestFunc(3, 4)          ' Prints 12
TestSub 3, 4                  ' Prints 12
```

TestFunc is a function that multiplies its two Integer arguments and returns the product as an Integer. TestSub multiplies its two Integer arguments and prints the product.

Declaring function and sub parameters

The parameters declared in a function or sub definition specify the types of the data to be passed as arguments to the function or sub when you invoke it.

Syntax

[**ByVal**] *argument* [**( )** | **List**] [**As** *dataType*]

The elements of a parameter declaration in a parameter list are described in the following table.

Element	*Description*
ByVal	Optional. A ByVal argument is passed by value. If this is omitted, the argument may be passed by reference.
argument	The name of the argument.
( ) | **List**	Optional. Parentheses mean the argument is an array variable. List means the argument is a list variable.
dataType	Optional. The data type of the argument.

ByVal means that the value assigned to *argument* when the function or sub is called is a local copy of a value in memory rather than a reference to that value. You can omit the clause As *dataType* and use a data type suffix character to declare the variable as one of the scalar data types. If you omit this clause and *argument* doesn't end in a data type suffix character (and isn't covered by an existing Def*type* statement), its data type is Variant.

Enclose the entire parameter list in parentheses, with a comma (,) following each parameter declaration except the last. The parameter list is written right after the function or sub name in the function or sub definition. (It may be preceded by white space.)

A function or sub can be defined with no parameters.

Examples: Declaring function and sub parameters

```
Function DCal (ByVal x As Single, y() As String, z$) As Integer
   ' ...
End Function
```

The function DCal takes three parameters:

- x, a Single argument that is passed by value
- y, a String array argument
- z, a String argument as declared by the $ data type suffix character

DCal returns an Integer value to the function caller.

Passing arguments by reference and by value

LotusScript provides two ways to pass arguments to functions and subs:

- By reference: A reference to the argument is passed. The function operates on the argument.
- By value: The value of the argument is copied into memory and the copy is passed. The function operates on the copy.

Whether an argument is passed by reference or by value depends on the data type and other characteristics of the argument:

- Arrays, lists, type instances, and objects must be passed by reference
- Constants and expressions are automatically passed by value
- Other arguments can be passed either by reference or by value, as specified in either the definition or the call of the function or sub. They are passed by reference unless the definition or the call specifies passing by value (see below).

Passing by reference

A variable passed to a function or sub by reference must have the same data type as the corresponding parameter in the function definition, unless the parameter is declared as Variant or is an object variable. An object variable can be passed to an object of the same class, an object of a base class, or an object of a derived class. In the latter case, the base class must contain an instance of the derived class or a class derived from the derived class.

A variable passed to a function or sub by reference, and modified by the function or sub, has the modified value when the function or sub returns.

Passing by value

LotusScript provides two ways for you to specify that a function or sub argument should be passed by value:

- Use the ByVal keyword in the argument's declaration in the function or sub definition.

  Use this method if you want the argument to be passed by value whenever the function or sub is called.

- Insert parentheses around the argument in the function or sub call.

  Use this method if you want to control whether an argument is passed by reference or by value at the time when the function or sub is called.

A value passed to a function or sub is automatically converted to the data type of the function or sub argument if conversion is possible. A Variant argument will accept a value of any built-in data type; and any list, array, or object.

If a variable is passed by value to a function or sub, and the argument is modified by the function or sub, the variable has its original value after the function or sub returns. The function or sub operates only on the passed copy of the variable, so the variable itself is unchanged.

Examples: Passing arguments by reference and by value

Example 1

```
' Define a function FOver with three Integer parameters:
' a variable, an array variable, and a list variable.
Function FOver(a As Integer, b() As Integer, c List As Integer)
   ' ...
End Function

Dim x As Integer, y As Integer
Dim y(5) As Integer
Dim z List As Integer

' Call the function FOver correctly, with arguments
' whose types match the types of the declared parameters.
Call FOver(x, y, z)
```

Example 2

```
' Define a function GLevel with one Integer list parameter.
Function GLevel (b List As Integer)
   ' ...
End Function

Dim z List As Integer

' Call the function GLevel incorrectly, passing a list
' argument by value.
Call GLevel ((z))
' Output:
' Error: Illegal pass by value: Z
' A list argument cannot be passed by value.
```

Example 3

```
' Define a function FExpr with two Integer parameters;
' the second must always be passed by value.
Function FExpr(a As Integer, ByVal b As Integer)
   ' ...
End Function

Dim x As Integer, w As Integer
Dim y(5) As Integer
Dim z List As Integer

' Call the function FExpr correctly with two Integer arguments:
' a constant and a variable.
```

```
Call FExpr(TRUE, x)
' Both arguments are passed by value:
' the first, TRUE, because it is a constant;
' and the second, x, because of the ByVal declaration in FExpr.

' The following call produces two error messages:
Call FExpr(TRUE, y)
' Output:
' Error: Illegal pass by value: Y
' Error: Type mismatch on: Y
' Because Y is an array variable, it is an illegal argument to pass by
' value and its type does not match the declared parameter type.
```

Example 4

```
' When a function modifies one of its parameters,
' the argument value is changed after the function returns if the
' argument was passed by reference. The value is not changed
' if the argument was passed by value.

Function FTRefOrVal(a As Integer) As Integer
   FTRefOrVal = a + 1
   a = a + 5
End Function

Dim x As Integer, y As Integer

' Show results of passing argument by reference.
Print x, FTRefOrVal(x As Integer), x
' Output:
' 0            1            5
' The value of x was changed from 0 to 5 in FTRefOrVal.

' Show results of calling with argument by value
' (note the extra parentheses around y%).
Print y, FTRefOrVal((y)), y
' Output:
' 0            1            0
' The value of the copy of y was changed from 0 to 5
' in FTRefOrVal. The value of y is unchanged.
```

Returning a value from a function

Within the body of a function definition, the function name automatically names a local variable. The data type of this variable is the same as the declared (or default) data type of the function's return value.

When the function is called, the variable is initialized to the LotusScript initial value for a variable of that data type. The variable value may be assigned one or more times

during execution of the function. The return value of the function is the value of the variable when the function returns from the function call.

Depending on how the function is called, the return value may be used or not after the function call returns.

Examples: Returning a value from a function

```
' Define the function TestFunc.

Function TestFunc (X As Integer, Y As Integer) As Integer
   TestFunc = X * Y
End Function

Dim holdVal As Integer
' Call the function TestFunc and use the return value.
holdVal = -1 + TestFunc(3, 4)      ' Set holdVal to 11.

' Call the function TestFunc and don't use the return value.
Call TestFunc 3, 4                 ' Return the value 12 (unused).
```

Recursive functions

A user-defined function can call itself. The function is called a recursive function. A call to itself from within the function is called a recursive call.

The definition of a recursive function must provide a way to end the recursion.

The depth of recursion is limited by available memory.

Examples: Recursive functions

Example 1

```
Public Function Factorial (n As Integer) As Long
   If n <= 1 Then               ' End the recursive calling chain.
      Factorial = 1
   Else
      Factorial = n * Factorial (n - 1)
   End If
End Function

Print Factorial(2)               ' Prints 2
Print Factorial(3)               ' Prints 6
Print Factorial(5)               ' Prints 120
```

Example 2

When recursively calling a function that has no arguments, you must insert empty parentheses following the function name in the call if you use the function's return value. The parentheses show that the function is being called. The function name without parentheses is interpreted as the variable that represents the return value of the function.

```
Function Recurse As Integer
    ' ...
    ' Call Recurse and assign the return value to x.
    x = Recurse()
    ' ...
    ' Assign the current value of the Recurse variable to x.
    x = Recurse
    ' ...
End Function
```

Chapter 3
Calling External C-Language Functions

LotusScript allows you to call external C language functions.

You implement external C functions inside a named library module. Under Windows®, this is a Dynamic Link Library (DLL). Such a library module generally contains several C functions.

In order to call C functions contained in an external library module from LotusScript, you must use the Declare statement for external C calls, for each function you want to call.

To avoid declaring external library functions in multiple scripts, use Declare Public statements in a module which remains loaded.

Under Windows 3.1, all external functions must use the Pascal calling convention. The following table shows the function calling convention that function calls from LotusScript to external functions must use on each LotusScript-supported platform.

Platform	*Calling convention*
Windows 3.1	Pascal
Windows 95, Windows NT™	STDCALL
OS/2®	_System
UNIX®	CDECL
Macintosh®	CDECL

If you are using Windows 95 or Windows NT, the name of an exported DLL function is case sensitive. However, LotusScript automatically converts the name to uppercase in the Declare statement. To successfully call an exported DLL, use the Alias clause in the Declare statement to specify the function name with correct capitalization. LotusScript leaves the alias alone.

Examples: Calling external C language functions

```
' The following statements declare an exported DLL with an alias
' (preserving case sensitivity), and then call that function.
Declare Function DirDLL Lib "C:\myxports.dll" _
   Alias "_HelpOut" (I1 As Long, I2 As Long)
DirDLL(5, 10)
```

Passing arguments to C functions

Arguments to C functions are passed either by reference or by value.

As described in the following, the rules for determining which types of data are legal arguments to C functions, and which arguments are passed by reference, and which passed by value, differ slightly from the rules for passing arguments to LotusScript functions and subs.

Passing arguments by value

When an argument is passed by value, the C function receives a copy of the actual value of the argument.

By default, arguments to C functions are passed by reference. To specify that an argument should be passed by value to a C function, you can use the same methods as for a LotusScript function or sub.

- To specify that the argument should always be passed by value, use the keyword ByVal preceding the parameter declaration for that argument in the Declare statement for the C function.
- To specify that the argument should be passed by value in a particular call to the function, use parentheses around the argument in the call.

The C routine cannot change the value of an argument passed by value, even if the C routine defines the argument as passed by reference.

The following table shows the data types that can be passed by value, and how the data is passed to the C function.

Data type	*How this data type is passed by value to a C function*
Integer	A 2-byte Integer value is pushed on the call stack.
Long	A 4-byte Long value is pushed on the call stack.
Single	A 4-byte Single value is pushed on the call stack.
Double	An 8-byte Double value is pushed on the call stack.
Currency	An 8-byte value, in the LotusScript internal Currency format, is pushed on the call stack.
String	A 4-byte pointer to the characters is pushed on the call stack. The C function should not write to memory beyond the end of the string. If the call is made with a variable, changes to the string by the C function will be reflected in the variable. Note that this is not true for a string argument to a LotusScript function declared as ByVal.
Variant	A 16-byte structure, in the LotusScript format for Variants, is pushed on the call stack.

continued

Data type	*How this data type is passed by value to a C function*
Product object	A 4-byte product object handle is pushed on the call stack.
Any	The number of bytes of data in the argument is pushed on the call stack. For example, if the argument contains a Long value, then the called function receives 4 bytes. **Note**: It is unknown at compile time how many bytes the function will receive at run time.

No other data types — arrays, lists, fixed-length strings, types, classes, or voids — can be passed by value. It is a run-time error to use these types as arguments.

Any of the data types that can be passed by value can also be passed by reference.

If a parameter is declared as type Any and the corresponding argument is passed by value, then any of the other data types listed in the preceding table can be used as that argument. The argument is passed as if it were specified as a ByVal argument of that other data type.

The argument ByVal &0 specifies a null pointer to a C function, when the argument is declared as Any.

Passing arguments by reference

When an argument is passed by reference, the C function receives a 4-byte pointer to the value area.

In some cases, the actual stack argument is changed to a publicly readable structure. In all cases, the data may be changed by the called function, and the changed value is reflected in LotusScript variables and in the properties of product objects. For such properties, this change occurs directly after the call has returned.

The following table lists the LotusScript data structures that can be passed by reference. The called C function receives a 4-byte pointer to these structures as the argument.

Data type	*How this data type is passed to a C function*
String	A 4-byte pointer to the string in the LotusScript internal string format.
Product object (including a collection)	A 4-byte product object handle.
Array	A 4-byte pointer to the array stored in the LotusScript internal array format.
Type	A 4-byte pointer to the data in the type instance. (This may include strings as elements.)
User-defined object	A 4-byte pointer to the data in the object. (This data may include strings, arrays, lists, product objects, etc., as elements.)

Note that a list cannot be passed either by value or by reference. A list is illegal as an argument to a C function.

Examples: Passing arguments to C functions

```
Declare Sub SemiCopy Lib "mylib.dll" _
   (valPtr As Integer, ByVal iVal As Integer)
Dim vTestA As Integer, vTestB As Integer
vTestA = 1
vTestB = 2

SemiCopy vTestA, vTestB

' The C function named SemiCopy receives a 4-byte pointer to a 2-byte
' integer containing the value of vTestA, and a 2-byte integer
' containing the value of vTestB.
' Since vTestA is passed by reference, SemiCopy can dereference the
' 4-byte pointer and assign any 2-byte integer to that location.

' When control returns to LotusScript, vTestA contains the modified
' value. Since vTestB was passed by value, any changes made by the
' C function are not reflected in vTestB after the function call.
```

String arguments to C functions

When a string is passed by reference, LotusScript passes a 4-byte pointer to a copy of the string in an internal buffer allocated in memory. The C function cannot safely modify the contents of this buffer unless the function is written specifically for LotusScript.

When a string is passed by value, LotusScript passes a 4-byte pointer to a Null-terminated string. The C function can modify the contents of this string, as long as it doesn't lengthen the string. Any changes to the string will be reflected in the script variable on return from the function.

You can specify the use of non-platform-native characters as arguments and return values using the LMBCS and Unicode keywords:

- Unicode specifies a Unicode string of two-byte characters (words) using the platform-native byte order.
- LMBCS specifies a LMBCS optimization group 1 string (multibyte characters).

For fixed-length strings passed by value, the string is first converted to a Null-terminated string. The C function can modify this string; but the string cannot be lengthened.

The following table summarizes the behavior of string arguments to a C function. It assumes a C function named cF with a string argument, declared by a Declare statement. The first column lists the ways you can declare the string argument of cF, while the second and third columns list the ways you can call cF.

Declaration for the string argument in the Declare statement for the C function cF	*How the arg is passed when cF is called in one of these forms:* *cF ((arg))* *cF (ByVal (arg))*	*How the arg is passed when cF is called in one of these forms:* *cF (arg)* *cF (ByVal arg)*
ByVal String	As a 4-byte pointer to a copy of *arg*'s character bytes. If cF changes the bytes, the value of *arg* does not change. If cF writes past the end of the string, it may be fatal to LotusScript.	As a 4-byte pointer to *arg*'s character bytes. If cF changes the bytes, the value of *arg* changes. If cF writes past the end of the string, it may be fatal to LotusScript.
String	As a 4-byte pointer to a copy of the string in the LotusScript internal string format. If cFchanges the bytes, the value of *arg* does not change.	As a 4-byte pointer to the string in the LotusScript internal string format. cF can change the bytes only by dereferencing the existing string and adding a reference to the new one.

Array, type, and object arguments to C functions

Passing arrays as arguments

Because LotusScript stores an array in a private format, you can pass an array by reference to a C function only if the function is specifically written for LotusScript. The following example shows how to declare and implement a C function that takes a LotusScript array of long values.

In LotusScript:

```
Declare Function LSArrayArg Lib "MYDLL" (ArrLng () As Long) As Long
Dim MyArr(0 to 5) As Long
Print LSArrayArg(MyArr)
```

In C:

```
long C_CALL_TYPE LSArrayArg(LSsValueArray *pLSArr)
{
     long *pData=pLSArr->Data; //pData points to first array element
     return pData[0]+pData[1]; //Sum first 2 array elements
}
```

Or:

```
long C_CALL_TYPE LSArrayArg(long **pLSArr)
{
     long *pData=*pLSArr; //pData points to first array element
     return pData[0]+pData[1]; //Sum first 2 array elements
```

C_CALL_TYPE is the calling convention: Pascal, STDCALL, _System, or CDEL.

Other C functions may require an array, such as the Windows function SetBitmapBits. You can still pass the array by passing the first array element by reference with the Any keyword, as shown in the following example.

In LotusScript:

```
Declare Function FncArrayArg(A As Any) As Long
Dim MyArr(0 to 5) As Long
Print FncArrayArg(MyArr(0))
```

In C:

```
long C_CALL_TYPE FncArrayArg(long *pArr)
{
     return pArr[0]+pArr[1]; //Sum first 2 array elements
}
```

Passing types as arguments

Some C functions can require a data structure as a parameter. An example is the Windows API function GetBrushOrgEx, which requires a pointer to a point structure. You can define a suitable data type, such as Point, and use that type definition to declare the C function. Since type variables are passed by reference, the C function receives a 4-byte pointer to the storage for the type variable.

LotusScript allows you to specify an optional string type, Unicode or LMBCS, on a type parameter in the Declare statement for a C function. The declarations have this form, for a function UniTest with one type argument and a function LMBCSTest with one type argument, where t1 is a user-defined data type:

```
Declare Function UniTest Lib "Unilib" (typArg As Unicode t1) As Long
Declare Function LMBCSTest Lib "lmbcslib" (typArg As LMBCS t1) As Long
```

In the first example, all strings (fixed- and variable-length) in type t1 and any of its nested types will be passed as Unicode strings. In the second example, all strings in type t1 (fixed- and variable-length) and any of its nested types will be passed as LMBCS strings.

If Unicode or LMBCS is not specified in this way, the default is to pass all strings in a type argument in the platform-native character set.

Strings contained in Variants in the type will not be affected.

Note that this change is incompatible with LotusScript Release 2, because translation to platform will be invoked by default on types containing strings (previously, these strings would have been passed as platform-native character set strings). This default behavior is compatible with LotusScript Release 2.

Also, in the case where the type contains a fixed-length non-Unicode string, the entire structure must be copied and its size adjusted. The size of the structure will be smaller (each fixed-length string will contain half as many bytes when translated to platform or LMBCS, since the length of the string is fixed and must be preserved). This implies that the string may be truncated (lose information) because a Unicode string of length 20 may require more than 20 bytes to represent in platform (DBCS). The loss cannot occur with variable-length strings, since they are represented as pointers.

LotusScript aligns type members to their natural boundaries for cross-platform transportability:

Data type	*Unicode*	*Platform*
Integer	2 bytes	2 bytes
Long	4 bytes	4 bytes
Single	4 bytes	4 bytes
Double	8 bytes	8 bytes
Currency	4 bytes	4 bytes
String	2 bytes	1 byte
Variant	8 bytes	8 bytes

The resulting alignment will not match the platform-specific alignment on Windows 3.1 and Windows 95. For example, consider this type definition:

```
Type telMet
   A As Integer
   B As Long
End Type
```

LotusScript will align the type member B on a 4-byte boundary, while the default alignment in Windows 3.1 will be on a 2-byte boundary.

Passing objects as arguments

When an object is passed to a C function, the function receives a 4-byte pointer to the unpacked data in the object. Because the data may include pointers to strings, arrays, lists, and product objects, it is unlikely that the C function has full knowledge of the internal structure of the object. You should use a C function to manipulate only the simplest objects.

Examples: Array, type, and object arguments to C functions

Example 1

```
' The following statements declare the C function SetBitmapBits.
' Its 3rd argument is an array argument. This is declared as type Any.
' In the function call, passing bitArray(0) passes the array
' by reference.
Declare Sub SetBitmapBits Lib "_privDispSys" _
   (ByVal hBM As Integer, ByVal cBytes As Long, pBits As Any)
' ...
SetBitmapBits(hBitmap, cBytesInArray, bitArray(0))
```

Example 2

```
' Define a Point type.
Type Point
   xP As Integer
   yP As Integer
End Type

' Call the C function GetBrushOrgEx with an argument of type Point.
Declare Function GetBrushOrgEx Lib "_pointLib" _
   (ByVal hDC As Integer, pt As Point) As Integer
Dim p As Point
' ...
GetBrushOrgEx(hDC,p)
```

Return values from C functions

The data type of a C function can be established by explicit data type declaration in the Declare statement, by a data type suffix character on the function name in the Declare statement, or by an applicable Def*type* statement. One of these must be in effect. Otherwise the data type of the return value defaults to Variant, which is illegal for a C function.

The following table shows which data types are legal as C function return value types.

Data type	*Legal as C function return type?*
Integer	Yes
Long	Yes
Single	Yes
Double	Yes
Currency	No
String	Yes, except for fixed-length string
Variant	No
Product object	Yes (as a 4-byte object handle of type Long)
User-defined object	Yes
Type instance	No
Any	No
Array	No
List	No

Chapter 4
File Handling

The following table describes the three kinds of files in LotusScript: sequential, random, and binary.

File type	*Description*
Sequential	The simplest and most common. It is the equivalent of a common text file. Data in sequential files is delimited with the platform's end-of-line indicator (carriage return, line feed, or both). The file is easily read with a text editor.
Random	The most useful for structured data. It is not readable except through LotusScript programs.
Binary	The most complex. It requires detailed programming to manipulate, because access is defined at the level of bytes on the disk.

To store and manipulate data in a file, the file must be opened first. When a file is opened in LotusScript, it is associated with a file number, an integer between 1 and 255. This number is used in most input and output operations to indicate which file is being manipulated. (A few file operations use the file name instead of a number.) The association between file name and file number remains until LotusScript closes the file.

Sequential files

A sequential file is an ordinary text file. Each character in the file is assumed to be either a text character or some other ASCII control character such as newline.

Sequential files provide access at the level of lines or strings of text: that is, data that is *not* divided into a series of records. However, a sequential file is not well suited for holding numbers, because a number in a sequential file is written as a character string.

Opening sequential files

A sequential file can be opened in one of three modes: input, output, or append. After opening a file, you must close it before opening it in another mode.

Use the Open statement in the following form to open a sequential file:

Open *fileName* [**For** {**Input** | **Output** | **Append**}] **As** *fileNumber* [**Len =** *bufferSize*]

Sequential input mode (using the keyword Input in the Open statement) signifies read-only access to the file, while Output signifies write-only. Append means write-only, but starting at the end of the file. Access in all three sequential modes is one line at a time.

The *bufferSize* supplied to the Open statement is the number of characters to be loaded into the internal buffer before being flushed to disk. This is a performance-enhancing feature: the larger the buffer, the faster the I/O. However, larger buffer sizes require more memory. The default buffer size for sequential files is 512 bytes.

When you try to open a file for sequential input, the file must already exist. If it doesn't, an error is generated. When you try to open a nonexistent file in output or append mode, the file is created automatically.

Writing to sequential files

To write the contents of variables to a sequential file (opened in output or append mode), use the Print # statement. For example:

```
Print #idFile, infV, supV
```

This writes the contents of infV and supV (separated by tabs, because of the commas in the statement) to the file numbered idFile. The parameters to Print can be strings or numeric expressions; they are converted to their string representations automatically.

The Write # statement also can write output to files. It generates output compatible with the Input # statement by separating each pair of expressions with a comma, and inserting quotation marks around strings. For example, consider these statements:

```
Dim supV As Variant, tailV As Variant
supV = 456
tailV = NULL
Write #idFile, "Testing", 123, supV, tailV
```

The statements generate the following line in the file numbered idFile:

```
"Testing",123,456,#NULL#
```

Reading from sequential files

To read data from a sequential file, open the file in input mode. Then use the Line Input # statement or the Input # statement, or the Input function, to read data from the file into variables.

Line Input # reads one line of text from a file, up to an end-of-line. The end-of-line is not returned in the string. For example:

```
Do Until EOF(idFile)
   Line Input #idFile, iLine
   Print iLine
Loop
```

These statements read a file one line at a time until end-of-file. The Print statement displays the line and appends an end-of-line sequence.

The Input # statement can be used to read in data that was formatted and written with the Write # statement. For example, suppose the file numbered idFile contained this line:

```
"Testing",123,456,#NULL#
```

Then the following statements would read “Testing” into liLabel, 123 into infA, 456 into supA, and the value NULL into tailV:

```
Dim liLabel As String, tailV As Variant
Dim infA As Integer, supA As Integer
Input #idFile, liLabel, infA, supA, tailV
```

If you find that you are using Write # and Input # with sequential files often, you should consider using a random file instead. Random files are better suited for record-oriented data.

You can also use the Input function to read data from a sequential file. This function takes the number of characters to read as an argument, and returns the characters. The Input$ function returns a string to the caller. The Input function returns a Variant variable.

For example, this example reads an entire file at once into a string variable:

```
Dim fulFile As String
fulFile = Input$(LOF(idFile), idFile)
' LOF returns the length of the file in characters.
```

Random files

A random file is made up of a series of records of identical length. A record can correspond to a scalar data type, such as Integer or String. Or it may correspond to a user-defined type, in which each record is broken down into fields corresponding to the members of the type.

The default mode for opening files is random mode. Thus, if you don't supply Binary or one of the sequential keywords (Input, Output, or Append), random mode is assumed.

Opening random files

Use the Open statement in the following form to open a random file:

Open *fileName* **For Random As** *fileNumber* [**Len =** *recordLength*]

Here, *recordLength* is the length of each record in the file. The default length is 128 bytes.

If the file does not exist, it is created.

Defining record types

Because records in a random file must all have the same length, strings in types should be fixed-length. Otherwise, data read from the file can be incorrect. If a string copied into a file record contains fewer characters than the record's fixed length, the remainder of the record is left unchanged. If a string is too long for a record, it is truncated when written.

For simplicity, string fields inside the user-defined type should be fixed-length. If you do use variable-length strings in your type, you must be sure that the Len part of the Open statement specifies a length that is large enough to hold the longest strings you wish to write to the file. Also, the Len function will not be able to give you a reliable value for the length of the record. (You will need to estimate that.) You will also be unable to navigate between records by omitting the record number in the Get and Put statements.

User-defined types can be used to define compound records. For example:

```
Type emploRec
   id As Integer                ' Integers are 2 bytes long
   salary As Currency           ' Currency is 8 bytes
   hireDate As Double           ' Dates are also 8 bytes
   lastName As String * 15      ' Fixed-length string of 30 bytes
```

```
   firstName As String * 15      ' Fixed-length string of 30 bytes
End Type
```

This record is 78 bytes long, so supply Len = 78 in the Open statement. The length of a type can be determined at run time using the Len function. For example:

```
Dim recLen As Integer, idFile As Integer
Dim recHold As emploRec
idFile = 1                      ' The file number to use for this
file
recLen = Len(recHold)           ' The record length for this file
Open "DATA.TXT" For Random As idFile Len = recLen
```

Reading from random files

Use the Get statement to read from a random file into variables. This example reads from the file numbered idFile, at record number 5, into the variable recHold:

```
' The record number to retrieve from the file
Dim posit As Integer
posit = 5
' The variable to read into
Dim recHold As emploRec
Get idFile, posit, recHold
```

Writing to random files

Use the Put statement to write to a random file. You can use Put to add new records to a random file, or to replace records in a random file.

Put takes three parameters: the file number to access, the record number to access, and a variable containing the data you wish to write. To replace a record in a random file, simply supply its record number:

```
Dim posit As Integer
posit = 5
' Replace record 5 with the contents of recHold.
Put idFile, posit, recHold
```

To add new records to a random file, use a record number equal to one more than the number of records in the file. To add a record to a file that contains 5 records, for example, use a position of 6.

To delete a record from a random file, you can move each record following it "down" by one position, thus overwriting the record you wish to delete. For example:

```
Dim tempRec As emploRec
For I = posit To lastRec - 1
   Get idFile, I + 1, tempRec
   Put idFile, I, tempRec
Next I
```

The problem with this technique is that it leaves a duplicate record at the end of the file. A better approach is to create a new file and copy all the valid records from the original file into the new file. Close the original file and then use the Kill statement to delete it, then use the Name statement to rename the new file to the same name as the original.

Binary files

Binary files are designed to provide the ultimate in control over the organization of your data for both reading and writing. The drawback is that you must know exactly how the file was written in order to read it properly. Though fewer functions and statements are available to manipulate data in binary files than in sequential files or random files, the binary file is also the most flexible.

Opening binary files

Use the Open statement in the following form to open a binary file:

Open *fileName* **For Binary As** *fileNumber*

If you include a record-length argument, it is ignored.

If the file does not exist, it is created, regardless of the access type supplied to the Open statement.

Using variable-length fields

Unlike random files, binary files can hold variable-length records. Thus, strings do not have to be fixed-length. However, you need to know the sizes of strings in order to read them. It is good programming practice to assign a length field to each variable-length record (each string) for this purpose. However, this is not necessary if the string is a component of a user-defined type. In this case, LotusScript automatically assigns a length field in the file to each variable-length string.

Binary access provides a byte-by-byte view of a file. A file appears to be a continuous stream of bytes, which may or may not be alphanumeric characters.

Writing to binary files

To write to a binary file, use the Put statement in this form:

Put *fileNumber*, *bytePosition*, *variableName*

The *bytePosition* parameter is the position in the file at which to start writing. The first byte in a file is at position 1; position zero is illegal, and specifying it results in an error.

Reading from binary files

To read data from a binary file, use either the Get statement or the Input function.

The Get statement reads the correct number of bytes into any variable of known length, such as a fixed-length string or an integer. For variable-length strings, the number of characters read equals the *current length* of the string. (Note that this will be zero for uninitialized variable-length strings.) Therefore you should first set the current length to the length of the string to be read. If the string in the file is within a user-defined type, the string length was stored by LotusScript with the string. If the string is not within a user-defined type, then you must know the length independently. For example, you can have stored the length as a separate field with the string.

The Input function or the Input$ function can also be used to read data from a binary file. The Input function is used in the form:

dataHold = **Input (***numBytes*, *fileNumber***)**

Here, dataHold is a Variant. (If the Input$ function were used instead of the Input function, dataHold should be a String.) This function reads *numBytes* bytes from the file and returns them in the variable dataHold.

Chapter 5
Error Processing

LotusScript recognizes two kinds of errors:

- Compiler errors are errors that are found and reported when the compiler attempts to compile the script. Common compiler errors are errors in the syntax of a statement, or errors in specifying the meaning of names.

  A compiler error prevents a script from being compiled. You need to correct any such errors by revising the script source statements that generated the error, and re-compiling the script, before the script can run.

- Run-time errors are found when LotusScript attempts to execute the script. A run-time error represents a condition that exists when the script is run, but could not be predicted at compile time. Examples of run-time errors are attempting to open a file that doesn't exist, or attempting to divide a number by a variable with a zero value.

  Run-time errors prevent a script from running to normal completion. When a run-time error occurs, script execution will end unless your script includes statements to handle the error.

LotusScript recognizes many run-time errors, and identifies each with a name, a number, and a standard message to describe it. Within a script, you can also define your own run-time errors and attach the same kind of information to them.

Defining errors and error numbers

Every error recognized at run time has its own error number that identifies it. When a recognized error happens during script execution, LotusScript records the error number, and then proceeds as directed by an On Error statement that refers to that number.

For example, you might write either one of these On Error statements to tell LotusScript how to respond to an occurrence of error number 357:

```
On Error 357 GoTo apoc600
On Error 357 Resume Next
```

Error numbers are established in two ways:

- By pre-definition in LotusScript

  LotusScript recognizes many common errors, and has a built-in error number associated with each one. The text file LSERR.LSS defines constants for each error; the value of the constant is the error number. To make these available to your script, include this file in your script with the statement:

  ```
  %Include "LSERR.LSS"
  ```

- By an Error statement in a script

  This statement signals error number 357:

  ```
  Error 357
  ```

  When this statement is executed, LotusScript records the occurrence of error 357 and then proceeds as directed by an On Error 357 statement. This facility has two uses:

  - You can use it to signal an error, give the error a number, and trigger error processing for that error. This is how you augment the pre-defined errors with errors and error processing specific to the needs of your script.
  - You can use it to simulate a pre-defined error. This is how you force LotusScript to execute some error-processing code without depending on the error to occur while other statements are executing. This is useful for testing the error-processing logic in your script.

When referring in an On Error statement to a pre-defined error, you can use the constant for the error, not the error number itself. For example, here are the statements in LSERR.LSS that define the error numbers and constants for two common errors:

```
Public Const ErrDivisionByZero      = 11 ' Division by zero
Public Const ErrIllegalFunctionCall = 5  ' Illegal function call
```

On Error statements then do not need to mention the numbers 11 and 5. Write the statements in this form instead, making the script easier to read:

```
On Error ErrDivisionByZero ...
On Error ErrIllegalFunctionCall ...
```

Similarly, you can define constants for your own error numbers. Then the constant names can be used instead of numbers in any Error statements and On Error statements that refer to the error. For example:

```
Const ooBounds = 677               ' A specific out-of-bounds error
' ...
Error ooBounds
' ...
On Error ooBounds ...
```

Run-time error processing

A run-time error occurs either when LotusScript executes an Error statement, or when executing some other statement results in an error. LotusScript then proceeds as follows.

The error is recorded as the current error. LotusScript records the line number in the script where the error occurred, the error number, and the error message associated with that number, if any. Until an error handling routine is invoked for this error, or another error is encountered, these are, respectively, the return values of the functions Erl, Err, and Error$. (Exception: The Err statement can be used to reset the current error number returned by the Err function.)

LotusScript then looks in the current procedure for an On Error statement associated with this error: either an On Error *n* statement, where *n* is the error number; or an On Error statement with no error number specified. If none is found, the search continues in the procedure that called this procedure, if any; and so on. If no associated On Error statement is found in any calling procedure, then execution ends and the associated error message is displayed.

If an associated On Error statement is found, then:

- If the On Error statement specifies Resume Next, execution continues in the procedure that contains the On Error statement. Execution continues with the statement following the statement that caused the error. (If the On Error statement is in a calling procedure, then the procedure call is considered to have caused the error; and all nested procedures are terminated before continuing execution.) The error is considered handled, but the return values of the functions Erl, Err, and Error$ are not reset.
- If the On Error statement specifies GoTo *label*, then execution continues at the statement labeled *label*. This must be in the same procedure as the On Error statement.

  In this case, the first Resume statement encountered after the labeled statement ends the error processing. The error is considered handled:

  - The return values of the functions Erl, Err, and Error$ are reset to their initial values: line number 0, error number 0, and the empty string (“”) as the error message, respectively
  - Execution continues at the location specified by the Resume statement.

Part 2
Language Elements

Chapter 6
Operators

The LotusScript operators are described in detail in the sections of this chapter. The operators are grouped according to the kinds of operations they perform.

Operator order of precedence

The simplest expression is a language element that represents a value: a constant literal, variable, function or property. You can build more complex expressions by combining subexpressions with operators.

The rules for determining the value of an expression are governed by the order in which the parts of an expression are evaluated. Operators with higher precedence are evaluated before operators with lower precedence. Operators with the same precedence are evaluated from left to right.

To override the normal order of evaluation in an expression, use parentheses. Subexpressions in parentheses are evaluated before the other parts of the expression, from left to right.

The following table summarizes the order of operator precedence. Operators on the same line have the same precedence.

LotusScript has both binary and unary operators. Binary operators take two operands; unary operators take one. The operands in the table are binary except where noted.

Operator	*Operation performed*
^	Exponentiation
- (unary)	Negation
* /	Multiplication and division
\	Integer division
Mod	Modulo division (remainder)
- +	Subtraction and addition
& +	String concatenation. To avoid ambiguity, use &. If an operand is a Variant, + can be interpreted as addition rather than concatenation.

continued

Operator	*Operation performed*
= <> >< < <= =< > >= =>	Numeric or string comparison
Like	Pattern matching. Same precedence as comparison operators.
Not (unary)	Logical negation (bit-wise)
And	Logical and (bit-wise)
Or	Logical or (bit-wise)
Xor	Logical exclusive-or (bit-wise)
Eqv	Logical equivalence (bit-wise)
Imp	Logical implication
Is	Object reference comparison

Examples: Operator order of precedence

```
' Arithmetic operators

Print 6 + 4 / 2                  ' Prints 8
Print (6 + 4) / 2                ' Prints 5

Print -2 ^ 2                     ' Prints -4
Print (-2) ^ 2                   ' Prints 4

' Comparison operators

Print 5 < 3                      ' Prints False
Print 5 > 3                      ' Prints True

Print "Hello" = "Hel" & "lo"     ' Prints True
```

Exponentiation operator

Raises a number to a power.

Syntax

number ^ *exponent*

Elements

number

Any numeric expression.

exponent

Any numeric expression. If *number* is negative, *exponent* must be an integer value.

Return value

The resulting data type is a Double or a Variant of type Double (DataType 5).

If either or both operands are NULL expressions, the result is a NULL.

Usage

Multiple ^ operators in a single expression are evaluated from left to right.

Examples: Exponentiation operator

```
Print 4 ^ 3       ' Prints 64
Print 4.5 ^ 3     ' Prints 91.125
Print -2 ^ 3      ' Prints -8
Print 2 ^ 3 ^ 2   ' Prints 64
' Use parentheses to change order of evaluation.
Print 2 ^ (3 ^ 2) ' Prints 512
```

Negation operator

Returns the negative value of a number.

Syntax

-*numExpr*

Elements

numExpr

Any numeric expression. An EMPTY operand (DataType 0) is considered a 0.

Return value

The result is of the same data type as *numExpr*. The data type of -*v*, where *v* has the value EMPTY, is Long.

If *numExpr* is a NULL, the result is a NULL.

Examples: Negation operator

```
Dim x As Integer
x% = 56
Print -x%               ' Prints -56
```

Multiplication operator

Multiplies two numbers.

Syntax

numExpr1 ***** *numExpr2*

Elements

numExpr1, *numExpr2*

Any numeric expressions. An EMPTY operand (DataType 0) is considered a 0.

Return value

The result is a value whose data type in most cases is the same as that of the operand whose data type is latest in this ordering: Integer, Long, Single, Double, Currency. For example, if one operand is a Double and the other is a Long, then the data type of the result is Double.

The exceptions are:

- If *numExpr1*, *numExpr2*, or both are NULL expressions, the result is a NULL.
- If *numExpr1* and *numExpr2* are both EMPTY, the result has DataType 2 (Integer).
- When the result has a Variant data type of DataType 3 (Long), DataType 4 (Single), or DataType 7 (Date/Time) that overflows its legal range, it's converted to a Variant of DataType 5 (Double). When the result has a Variant of DataType 2 (Integer) that overflows its legal range, it's converted to a Variant of DataType 3 (Long).

Examples: Multiplication operator

```
Dim x As Integer
x% = 2 * 3
Print x% * 3.4              ' Prints 20.4
```

Division operator

Divides two numbers and returns a floating-point result.

Syntax

numExpr1 / *numExpr2*

Elements

numExpr1, *numExpr2*

Any numeric expressions. An EMPTY operand (DataType 0) is considered a 0.

Return value

The resulting data type is a Double or a Variant of DataType 5 (Double).

If either or both operands are NULL expressions, the result is a NULL.

Examples: Division operator

This example contrasts ordinary division with integer division. Integer division rounds, then divides, and then drops the fractional part. Note that because the operands are rounded before division, the result may differ from the integer part of an ordinary division operation.

```
Print 8 / 5                    ' Prints 1.6
Print 8 \ 5                    ' Prints 1
Print 16.9 / 5.6               ' Prints 3.01785714285714
Print 16.9 \ 5.6               ' Prints 2
```

Integer division operator

Performs integer division on two numbers and returns the result.

Syntax

numExpr1 \ *numExpr2*

Elements

numExpr1, *numExpr2*

Any numeric expressions. An EMPTY operand (DataType 0) is considered a 0.

Return value

The result is of data type Integer, Long, or Variant of type Integer (DataType 2) or Long (DataType 3).

If either or both operands are NULL expressions, the result is a NULL.

Usage

LotusScript rounds the value of each operand to an Integer or Long value. Then *numExpr1* is divided by *numExpr2* as an ordinary numerical division; and any fractional part of the result is dropped.

Examples: Integer division operator

This example contrasts ordinary division with integer division. Integer division rounds, then divides, and then drops the fractional part. Note that because the operands are rounded before division, the result may differ from the integer part of an ordinary division operation.

```
Print 8 / 5                    ' Prints 1.6
Print 8 \ 5                    ' Prints 1
Print 16.9 / 5.6               ' Prints 3.01785714285714
Print 16.9 \ 5.6               ' Prints 2
```

Mod operator

Divides two numbers and returns the remainder.

Syntax

numExpr1 **Mod** *numExpr2*

Elements

numExpr1, *numExpr2*

Any numeric expressions. An EMPTY operand (DataType 0) is considered a 0.

Return value

The result is of data type Integer, Long, or Variant of type Integer (DataType 2) or Long (DataType 3).

If either or both operands are NULL expressions, the result is a NULL.

Usage

The remainder operator divides *numExpr1* by *numExpr2* and returns the remainder.

The operands are rounded to Integer expressions before the division takes place.

Examples: Mod operator

```
Print 17 Mod 3                  ' Prints 2
Print 16.9 \ 5.6                ' Prints 5
```

Addition operator

Adds two numbers.

Syntax

numExpr1 **+** *numExpr2*

Elements

numExpr1, *numExpr2*

Any numeric expressions. An EMPTY operand (DataType 0) is considered a 0.

Return value

When adding expressions of numeric data types, the result is a value whose data type in most cases is the same as that of the operand whose data type is latest in this ordering: Integer, Long, Single, Double, Currency. For example, if one operand is a Double and the other is an Integer, then the data type of the result is Double.

The exceptions are:

- When the resulting data type is a Variant of DataType 2 (Integer) that overflows its legal range, the result is converted to a Variant of DataType 3 (Long).
- If *numExpr1* and *numExpr2* are both EMPTY, the result has DataType 2 (Integer).
- When the resulting data type is a Variant of DataType 3 (Long), DataType 4 (Single), or DataType 7 (Date/Time) that overflows its legal range, the result is converted to a Variant of DataType 5 (Double).

Usage

LotusScript interprets the + operator as either addition or string concatenation, depending on the data types of *expr1* and *expr2*. The following table lists these interpretations. The numeric data types are Integer, Long, Single, Double, Currency, and (in a Variant variable only) Date/Time.

One expression	*Other expression*	*Operation*
Numeric	Numeric	Addition
Numeric	String	(Type mismatch error occurs)
Numeric	Variant (other than NULL)	Addition
String	Variant (other than NULL)	String concatenation
String	String	String concatenation
Any type	Variant that contains EMPTY	Returns first expression
Any type	NULL	Returns NULL
Variant of numeric data type	Variant of numeric data type	Addition
Variant of numeric data type	Variant of String data type	Addition
Variant of String data type	Variant of String data type	String concatenation

To avoid confusion, you should use the & operator, not the + operator, for string concatenation.

Examples: Addition operator

```
Dim a As Variant
Dim b As Integer
a = "8"
b% = 7

' Use operator for addition.
Print 8 + 7                          ' Prints 15
Print a + 7                          ' Prints 15
Print 8 + b%                         ' Prints 15
Print a + b%                         ' Prints 15

' Use operator for string concatenation.
Print "Hello " + "there"              ' Prints "Hello there"
Print a + "7"                        ' Prints "87"
```

Subtraction operator

Finds the difference between two numbers.

Syntax

numExpr1 - numExpr2

Elements

numExpr1, *numExpr2*

Any numeric constant, variable, or expression; or any function that returns a number. An EMPTY operand (DataType 0) is considered a 0.

Return value

The result is a value whose data type in most cases is the same as that of the operand whose data type is latest in this ordering: Integer, Long, Single, Double, Currency. For example, if one operand is a Long and the other is a Currency, then the data type of the result is Currency.

The exceptions are:

- When the result is a Variant of DataType 2 (Integer) that overflows its legal range, the result is converted to a Variant of DataType 3 (Long).
- When the result is a Variant of DataType 3 (Long), DataType 4 (Single), or DataType 7 (Date/Time) that overflows its legal range, the result is converted to a Variant of DataType 5 (Double).
- If *numExpr1* and *numExpr2* are both EMPTY, the result has DataType 2 (Integer).
- If either or both operands are NULL expressions, the result is a NULL.

Examples: Subtraction operator

```
Print 5 - 3.4          ' Prints 1.6
```

Comparison operators

Compare two expressions.

Syntax

expr1 operator expr2

Elements

expr1, expr2

Any expressions.

operator

One of the following operators: <, >, <=, =<, >=, =>, <>, ><, =.

Return value

The result of a comparison is TRUE or FALSE. If either or both operands are NULL expressions, the result is a NULL.

The following table lists the results of comparing two expressions, neither of which is NULL.

Operator	*Operation*	*TRUE if*	*FALSE if*
<	Less than	*expr1* < *expr2*	*expr1* >= *expr2*
<= or =<	Less than or equal to	*expr1* <= *expr2*	*expr1* > *expr2*
>	Greater than	*expr1* > *expr2*	*expr1* <= *expr2*
>= or =>	Greater than or equal to	*expr1* >= *expr2*	*expr1* < *expr2*
=	Equal to	*expr1* = *expr2*	*expr1* <> *expr2*
<> or ><	Not equal to	*expr1* <> *expr2*	*expr1* = *expr2*

Usage

Comparison operators are also called relational operators.

LotusScript interprets the comparison operator as either numeric comparison or string comparison, depending on the data types of *expr1* and *expr2*. The following table lists these interpretations. The numeric data types are Integer, Long, Single, Double, Currency, and (in a Variant variable only) Date/Time.

One expression	*Other expression*	*Operation*
Numeric	Numeric	Numeric comparison
Numeric	Variant of numeric data type or Variant containing string value that can be converted to a number	Numeric comparison

continued

One expression	*Other expression*	*Operation*
Numeric	Variant containing String value that cannot be converted to a number	Type mismatch error occurs.
Numeric	Variant that contains EMPTY	Numeric comparison, with 0 substituted for the EMPTY expression.
String	String	String comparison
String	Variant (other than NULL)	String comparison
String	Variant that contains EMPTY	String comparison
Variant containing string value	Variant containing string value	String comparison
Variant that contains EMPTY	Variant containing string value	String comparison, with the empty string (“”) substituted for the EMPTY expression.
Variant of numeric data type	Variant of numeric data type	Numeric comparison
Variant that contains EMPTY	Variant of numeric data type	Numeric comparison, with 0 substituted for the EMPTY expression.
Variant of numeric data type	Variant containing string value	Numeric comparison. The numeric expression is less than the string expression.
Variant that contains EMPTY	Variant that contains EMPTY	Expressions are equal.

For string comparison, the Option Compare statement sets the comparison method:

- Option Compare Case and Option Compare NoCase specify comparison using the character collating sequence determined by the Lotus product that you are using. Case specifies case sensitive comparison, and NoCase specifies case insensitive comparison.
- Option Compare Pitch and Option Compare NoPitch specify comparison using the character collating sequence determined by the Lotus product that you are using. Pitch specifies pitch sensitive comparison, and NoPitch specifies pitch insensitive comparison. These options apply to Asian (double byte) characters.
- Option Compare Binary specifies string comparison in the platform’s collating sequence. The effect is platform sort-order, case sensitive comparison.

If you omit the Option Compare statement, the default method of string comparison is the same as Option Compare Case, Pitch.

To compare strings, LotusScript examines the two strings character by character, starting with the first character in each string. The collating sequence values (positions in the character sort sequence) of the two characters are compared.

- If these values are not equal, the string whose character has the larger collating sequence value (appears later in the sort sequence) is the larger string.
- If the collating sequence values of the pair of characters are the same, and both strings contain more characters, then the character comparison proceeds to the next character.

If the end of both strings is reached simultaneously by this process, then neither string has been found larger than the other, and the strings are equal. Otherwise the longer string is the larger string.

Examples: Comparison operators

```
' Use operator for numeric comparisons.
Print 1 < 2                              ' Prints True
Print 2 > 1                              ' Prints True
Print 1 <> 2                             ' Prints True
Print 2 >= 2                             ' Prints True
Print 2 <= 2                             ' Prints True
Print 2 = 2                              ' Prints True

' Use operator for string comparisons.
Print "hello" < "hellp"                   ' Prints True

Dim myVar As Variant, myStr As Variant
myStr = "34"
myVar = 34
Print myVar < myStr                      ' Prints True
Print 45 > myStr                         ' Prints True
Print "45" > myVar                       ' Prints True
```

Not operator

Performs logical negation on an expression. The Not operator has the effect of rounding its argument to the nearest integer, changing the sign, and subtracting 1.

Syntax

Not *expr*

Elements

expr

Any expression. Its value must lie within the range for Long values.

Usage

The following table explains how LotusScript determines the result of the Not operation.

expr	*Result*
TRUE	FALSE
FALSE	TRUE
NULL	NULL

In addition to performing logical negation, the Not operator reverses the bit values of any variable and sets the corresponding bit in the result according to the following table.

Bit n in expr	*Bit n in result*
0	1
1	0

Examples: Not operator

```
Print Not TRUE                    ' Prints False
Print Not 12.4                    ' Prints -13
```

And operator

Performs a logical conjunction on two expressions. LotusScript rounds to the nearest integer before performing the And operation.

Syntax

expr1 **And** *expr2*

Elements

expr1, *expr2*

Any expressions. Their values must lie within the range for Long values.

Usage

The following table explains how LotusScript determines the result when the And operator is used.

expr1	*expr2*	*Result*
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE
TRUE	NULL	NULL
NULL	TRUE	NULL
FALSE	NULL	FALSE
NULL	FALSE	FALSE
NULL	NULL	NULL

In addition to performing a logical conjunction, the And operator compares identically positioned bits in two numeric expressions (known as a bit-wise comparison) and sets the corresponding bit in the result as follows.

Bit n in expr1	*Bit n in expr2*	*Bit n in result*
1	1	1
1	0	0
0	1	0
0	0	0

Examples: And operator

```
' Boolean usage
Dim johnIsHere As Integer, jimIsHere As Integer
Dim bothAreHere As Integer
johnIsHere% = TRUE
jimIsHere% = FALSE
bothAreHere% = johnIsHere And jimIsHere
Print bothAreHere%                  ' Prints 0 (False)

' Bit-wise usage
Dim x As Integer, y As Integer
x% = &b11110000
y% = &b11001100
Print Bin$(x% And y%)               ' Prints 11000000
```

Or operator

Performs a logical disjunction on two expressions.

Syntax

expr1 **Or** *expr2*

Elements

expr1, *expr2*

Any expressions. Their values must lie within the range for Long values.

Usage

The following table explains how LotusScript determines the result of the Or operation.

expr1	*expr2*	*Result*
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE
TRUE	NULL	TRUE
NULL	TRUE	TRUE
FALSE	NULL	NULL
NULL	FALSE	NULL
NULL	NULL	NULL

In addition to performing a logical disjunction, the Or operator compares identically positioned bits in two numeric expressions (known as a bit-wise comparison) and sets the corresponding bit in the result according to the following table.

Bit n in expr1	*Bit n in expr2*	*Bit n in result*
1	1	1
1	0	1
0	1	1
0	0	0

Examples: Or operator

```
' Boolean usage
Dim johnIsHere As Integer, jimIsHere As Integer
Dim oneOrMoreIsHere As Integer
johnIsHere% = TRUE
jimIsHere% = FALSE
oneOrMoreIsHere% = johnIsHere% Or jimIsHere%
Print oneOrMoreIsHere%                ' Prints -1 (True)

' Bit-wise usage
Dim x As Integer, y As Integer
x% = &b11110000
y% = &b11001100
Print Bin$(x% Or y%)                  ' Prints 11111100
```

Xor operator

Performs a logical exclusion on two expressions.

Syntax

expr1 **Xor** *expr2*

Elements

expr1, *expr2*

Any expressions. Their values must lie within the range for Long values.

Usage

The following table explains how LotusScript determines the result of the Xor operation.

expr1	*expr2*	*Result*
TRUE	TRUE	FALSE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE
TRUE	NULL	NULL
NULL	TRUE	NULL
FALSE	NULL	NULL
NULL	FALSE	NULL
NULL	NULL	NULL

In addition to performing a logical exclusion, the Xor operator compares identically positioned bits in two numeric expressions (known as a bit-wise comparison) and sets the corresponding bit in the result according to the following table.

Bit n in expr1	*Bit n in expr2*	*Bit n in result*
1	1	0
1	0	1
0	1	1
0	0	0

Examples: Xor operator

```
' Boolean usage
Dim johnIsHere As Integer, jimIsHere As Integer
Dim oneButNotBothIsHere As Integer
johnIsHere% = TRUE
jimIsHere% = FALSE
oneButNotBothIsHere% = johnIsHere% Xor jimIsHere%
Print oneButNotBothIsHere%         ' Prints -1 (True)

' Bit-wise usage
Dim z As Integer, y As Integer
z% = &b11110000
y% = &b11001100
Print Bin$(z% Xor y%)              ' Prints 111100
```

Eqv operator

Performs a logical equivalence on two expressions.

Syntax

expr1 **Eqv** *expr2*

Elements

expr1, *expr2*

Any expressions. Their values must lie within the range for Long values.

Usage

The following table explains how LotusScript determines the result of the Eqv operation.

expr1	*expr2*	*Result*
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	TRUE
TRUE	NULL	NULL
NULL	TRUE	NULL
FALSE	NULL	NULL
NULL	FALSE	NULL
NULL	NULL	NULL

In addition to performing a logical equivalence, the Eqv operator compares identically positioned bits in two numeric expressions (known as a bit-wise comparison) and sets the corresponding bit in the result according to the following table.

Bit n in expr1	*Bit n in expr2*	*Bit n in result*
1	1	1
1	0	0
0	1	0
0	0	1

Examples: Eqv operator

```
Dim a As Variant, b As Variant, c As Variant
a = &HF
b = &HF0
c = &H33
Print TRUE Eqv TRUE              ' Prints True
Print FALSE Eqv FALSE            ' Prints True
Print TRUE Eqv FALSE             ' Prints False
Print Hex$(a Eqv b)              ' Prints FFFFFF00
Print Hex$(a Eqv c)              ' Prints FFFFFFC3
Print Hex$(b Eqv c)              ' Prints FFFFFF3C
```

Imp operator

Performs a logical implication on two expressions.

Syntax

expr1 **Imp** *expr2*

Elements

expr1, *expr2*

Any expressions. Their values must lie within the range for Long values.

Usage

The following table explains how LotusScript determines the result of the Imp operation.

expr1	*expr2*	*Result*
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	TRUE
FALSE	FALSE	TRUE
TRUE	NULL	NULL
NULL	TRUE	TRUE
FALSE	NULL	TRUE
NULL	FALSE	NULL
NULL	NULL	NULL

In addition to performing a logical implication, the Imp operator compares identically positioned bits in two numeric expressions (known as a bit-wise comparison) and sets the corresponding bit in the result according to the following table.

Bit n in expr1	*Bit n in expr2*	*Bit n in result*
1	1	1
1	0	0
0	1	1
0	0	1

Examples: Imp operator

```
Dim youCanSee As Integer, lightIsOn As Integer

' You don't need the light to see.
youCanSee% = TRUE
lightIsOn% = FALSE
Print youCanSee% Imp lightIsOn%          ' Prints False

' You need the light to see.
youCanSee% = FALSE
lightIsOn% = FALSE
Print youCanSee% Imp lightIsOn%          ' Prints True
```

String concatenation operator

Concatenates two expressions as strings.

Syntax

expr1 **&** *expr2*

Elements

expr1, *expr2*

Any String expressions, or any of the following:

- Numeric expression: LotusScript converts it to a Variant of DataType 8 (String).
- NULL: LotusScript treats it as a zero-length String value when concatenated with the other expression. If both expressions are NULL, the result is NULL.
- EMPTY: LotusScript treats it as a zero-length String value.

Return value

The result is a Variant of DataType 8 (String).

Usage

The + operator also concatenates two character strings, but you should use the & operator for string concatenation to avoid confusion.

Examples: String concatenation operator

```
Dim x As Variant
x = 56 & " Baker St."
Print x                          ' Prints "56 Baker St."
```

Like operator

Determines whether a string expression matches a pattern string.

Syntax

stringExpr **Like** *patternString*

Elements

stringExpr

Any String expression.

patternString

A string expression that can include any individual ANSI characters and any of the wildcard characters or collections that follow. You can use as many wildcards as you need within a single *patternString*.

Wildcard	*Matches*
?	Any one character
#	Any one digit from 0 through 9
*****	Any number of characters (zero or more)
[*characters***]**	Any one of the *characters* in the list or range specified here
[!*characters***]**	Any one character not included in the list or range of *characters* specified here

Usage

Matching characters in a list

For a list, just type the characters; don't use any type of separator, not even a space (the separator would be considered part of the list). For example, [1 2 3 4 5] represents the characters 1, space, 2, 3, 4, and 5 (the redundant occurrences of the space would be ignored). But [12345] represents the characters 1, 2, 3, 4, and 5 (with no space character).

Matching characters in a range

For a range, separate the lower and upper bounds with a hyphen, as in [1-5].

Specify the range in ascending character collating sequence order (for example, [5-1] is invalid). If binary comparison (Option Compare Binary) is in effect, LotusScript uses the international ANSI character collating sequence. This is the sequence of values Chr(0), Chr(1), Chr(2), ... It is the same on all LotusScript platforms. A range specified in ascending order will produce a valid pattern string. However, if Option Compare Case, NoCase, Pitch, or NoPitch is in effect, then the collating sequence order depends on the Lotus product that you are using. The order for alphanumeric characters will be the same as international ANSI, but using other characters to define a range may produce an invalid pattern string. If you define a range using nonalphanumeric

characters, specify Option Compare Binary in your script to be certain of defining a valid pattern string.

When you specify multiple ranges, you don't have to separate them with anything; for example, [1-5A-C] contains the ranges 1-5 and uppercase A-C.

Matching special characters

To match one of these characters, include it in a *characters* list:

- Hyphen (**-**)
- Question mark (**?**)
- Asterisk (*****)
- Number sign (**#**)
- Open bracket (**[**)

Be sure to place the hyphen at the beginning of the list; if you're using the [!*characters*] format, the hyphen immediately follows the exclamation point, as in [!-123]. The other characters can appear anywhere in the *characters* list. For example, [-?A-Z] matches the hyphen, the question mark, and any uppercase letter from A through Z.

To match one of these characters, place the character anywhere within your wildcard specification except in a *characters* list or range:

- Comma (**,**)
- Close bracket (**]**)
- Exclamation mark (**!**)

For example, !,[1-6] matches the exclamation point, the comma, and any digit from 1 through 6.

Return value

If *stringExpr* matches *patternString*, the result is TRUE; if not, the result is FALSE. If either *stringExpr* or *patternString* is NULL, the result is NULL.

Usage

By default, the comparison is case sensitive. You can modify case sensitivity with the Option Compare statement.

Examples: Like operator

Example 1

```
' Print the numbers from 1 to 100 that include the digit 5.
For x = 1 To 100
   If CStr(x) Like "*5*" Then Print x
Next x
' Output:
' 5 15 25 35 45 50 51 52 53 54 55 56 57 58 59 65 75 85 95
```

Example 2

```
' Print the numbers from 1 to 100 that end in 3 and don't begin with
2.
For x = 1 To 100
   If CStr(x) Like "[!2]3" Then Print x
Next x
' Output:
' 13 33 43 53 63 73 83 93
```

Is operator

Compares two object reference variables.

Syntax

obj1 **Is** *obj2*

Elements

obj1, *obj2*

Expressions whose values are object references.

Usage

The result of the Is operator is TRUE only if *obj1* and *obj2* refer to the same object.

The operands *obj1* and *obj2* may be Variant variables, object reference variables, or any variable elements that accept an object reference, such as an element of an array, list, or user-defined type.

Examples: Is operator

```
Class MyClass
   ' ...
End Class
Dim x As MyClass
Dim y As MyClass
Set x = New MyClass
Set y = New MyClass
Print x Is y                    ' Prints False
Set x = y                       ' x now refers to the same object as
y.
Print x Is y                    ' Prints True
```

IsA operator

Determines if an object reference variable is of a specified class or a class derived from the specified class.

Syntax

obj **IsA** *objName*

Elements

obj

Expression whose value is an object reference.

objName

String expression representing an object name.

Usage

The result of the IsA operator is TRUE if *obj* is of the class *objName* or a class derived from *objName.*

Obj can be a native (user defined) object, a product object, or an OLE object.

Obj can be a Variant variable, an object reference variable, or any variable element that accepts an object reference, such as an element of an array, list, or user-defined type or class. *Obj* can be an expression, for example, a function that returns an object or array of objects.

ObjName represents the class that is visible in the current scope if the class name defines more than one class.

Examples: IsA operator

```
Sub PrintIt(objA)
     If objA IsA "ClassA" Then
          objA.Print
     Else
          Print "Not a ClassA object"
     End If
End Sub
```

Chapter 7
Statements, Built-In Functions, Subs, Data Types, and Directives

Abs function

Returns the absolute value of a number.

Syntax
Abs (*numExpr* **)**

Elements
numExpr
Any numeric expression.

Return value
Abs returns the absolute value of *numExpr*.

The data type of the return value is the same as the data type of *numExpr*, unless *numExpr* is a Variant. In that case, the following rules apply:

- If *numExpr* contains a string that LotusScript can convert to a number, the data type is Double.
- If *numExpr* contains a value that LotusScript cannot convert to a number, the function returns an error.
- If *numExpr* contains a NULL, the return value is NULL.

Usage
The absolute value of a number is its unsigned magnitude; for example, 3 and -3 both have an absolute value of 3.

Examples: Abs function
```
Print Abs(12)                          ' Prints 12
Print Abs(-12)                         ' Prints 12
Print Abs(13 - 25)                     ' Prints 12
Print TypeName(Abs(-12))               ' Prints INTEGER

Dim someV As Variant
someV = "123"
Print Abs(someV)                       ' Prints 123
someV = NULL
Print Abs(someV)                       ' Prints #NULL#
```

ACos function

Returns the arccosine, in radians, of a number between -1 and 1, inclusive.

Syntax

ACos (*numExpr* **)**

Elements

numExpr

A numeric expression with a value between -1 and 1, inclusive.

Return value

ACos returns the arc cosine, in radians, of the value of *numExpr*.

The range of the return value is zero to PI, inclusive.

The data type of the return value is Double.

If the value of *numExpr* is not in the range -1 to 1, inclusive, the function returns an error.

Usage

The arc cosine of a number is the angle, in radians, whose cosine is equal to the value of that number.

Examples: ACos function

```
Dim rad As Double
Dim degrees As Double

' Assign the value PI/2, the angle whose cosine is 0.
rad# = ACos(0)

' Assign the value 90, the same angle in degrees.
degrees# = rad# * (180 / PI)

Print rad#; degrees#                    ' Prints 1.5707963267949  90
```

ActivateApp statement

Makes a program window the active window.

Syntax

ActivateApp *windowName*

AppActivate is acceptable in place of ActivateApp.

Elements

windowName

A string expression designating the program window to activate.

Usage

windowName is not case-sensitive. It must exactly match the leftmost characters of the program title that appears in the program window title bar. For example, if the program title of a running program window is "Lotus Notes - Workspace", then a *windowName* value of "Lotus Notes" will activate that window. If more than one program title matches *windowName*, LotusScript will choose one of the program windows.

ActivateApp can activate a minimized window, but cannot restore or maximize it. Use SendKeys to restore or maximize a window. Use Shell to start a program.

Examples: ActivateApp statement

```
' Activate the Lotus Notes program window
' (assuming that Lotus Notes is already running).
' This would match a windows with the title "Lotus Notes - Workspace".
ActivateApp "Lotus Notes"
```

Asc function

Returns the platform-specific numeric character code for the first character in a string.

Syntax

Asc (*stringExpr* **)**

Elements

stringExpr

Any string expression.

Return value

Asc returns the numeric character code of the first character in *stringExpr*. The code represents the character in the character set of the platform on which you are running LotusScript.

The data type of the return value is Long.

If the value of *stringExpr* is NULL or the empty string (""), the function returns an error.

Examples: Asc function

```
Dim bigA As Long
Dim littleA As Long
bigA& = Asc("A")
littleA& = Asc("a")
Print bigA&; littleA&                    ' Prints 65  97
```

ASin function

Returns the arcsine, in radians, of a number between -1 and 1, inclusive.

Syntax

ASin (*numExpr* **)**

Elements

numExpr

A numeric expression with a value between -1 and 1, inclusive.

Return value

ASin returns the angle, in radians, whose sine is equal to the value of *numExpr*.

The range of the return value is -PI/2 to PI/2, inclusive.

The data type of the return value is Double.

If the value of *numExpr* is not in the range -1 to 1, inclusive, the function returns an error.

Examples: ASin function

```
Dim rad As Double
Dim degrees As Double

' Assign the value PI/2, the angle whose sine is 1.
rad# = ASin(1)

' Assign the value 90, the same angle in degrees.
degrees# = rad# * (180 / PI)

Print rad#, degrees#                  ' Prints  1.5707963267949  90
```

ATn function

Returns the arctangent, in radians, of a number.

Syntax

ATn (*numExpr* **)**

Elements

numExpr

Any numeric expression.

Return value

ATn returns the angle, in radians, whose tangent is equal to the value of *numExpr*.

The range of the return value is -PI/2 (-90 degrees) to PI/2 (90 degrees), exclusive.

The data type of the return value is Double.

Examples: ATn function

```
Dim rad As Double
Dim degrees As Double

' Assign the value PI/4, the angle whose tangent is 1.
rad# = ATn(1)

' Assign the value 45, the same angle in degrees.
degrees# = rad# * (180 / PI)

Print rad#; degrees#                    ' Prints .785398163397449  45
```

ATn2 function

Returns the polar coordinate angle, in radians, of a point in the Cartesian plane.

Syntax

ATn2 (*numExprX* **,** *numExprY* **)**

Elements

numExprX, *numExprY*

Any numeric expressions. At least one of the two must be non-zero. *numExprX* and *numExprY* designate the coordinates of a point in the Cartesian plane.

Return value

ATn2 returns the angular portion of the polar coordinate representation of the point *(numExprX, numExprY)* in the Cartesian plane.

The range of the return value is -PI to PI, inclusive.

If *numExprX* is 0, then ATn2 returns one of the following values:

- -PI/2, if *numExprY* is negative
- PI/2, if *numExprY* is positive

If *numExprX* is positive, then ATn2(*numExprX, numExprY*) returns the same value as ATn(*numExprY / numExprX*).

Examples: ATn2 function

```
Dim quad1 As Double, quad2 As Double, _
   quad3 As Double, quad4 As Double

' Assign the arctangents of four points in the plane.
quad1# = ATn2(1, 1)
quad2# = ATn2(-1, 1)
quad3# = ATn2(-1, -1)
quad4# = ATn2(1, -1)

' Print the value each angle in degrees.
Print quad1# * (180 / PI)              ' Prints 45
Print quad2# * (180 / PI)              ' Prints 135
Print quad3# * (180 / PI)              ' Prints -135
Print quad4# * (180 / PI)              ' Prints -45
```

Beep statement

Generates a tone on the computer.

Syntax

Beep

Usage

The tone that LotusScript produces depends on the sound-generating hardware in your computer.

Examples: Beep statement

```
' While a user-specified interval (in seconds) is elapsing, beep and
' count the beeps. Then tell the user the number of beeps.

Dim howLong As Single, howManyBeeps As Integer
Function HowManyTimes (howLong As Single) As Integer
   Dim start As Single, finish As Single, counter As Integer
   start! = Timer
   finish! = start! + howLong!
   While Timer < finish!
      Beep
      counter% = counter% + 1
   Wend
   HowManyTimes% = counter%
End Function
howLong! = CSng(InputBox _
   ("For your own sake, enter a small number."))
howManyBeeps% = HowManyTimes(howLong!)
MessageBox "Number of beeps:" & Str(howManyBeeps%)
```

Bin function

Returns the binary representation of a number as a string.

Syntax

Bin[**$**] **(** *numExpr* **)**

Elements

numExpr

Any numeric expression. If *numExpr* evaluates to a number with a fractional part, LotusScript rounds it to the nearest integer before deriving its binary representation.

Return value

Bin returns a Variant of DataType 8 (String), and Bin$ returns a String.

Return values will only include the characters 0 and 1. The maximum length of the return value is 32 characters.

Usage

If the data type of *numExpr* is not Integer or Long, then LotusScript attempts to convert it to a Long. If it cannot be converted, a type mismatch error occurs.

Examples: Bin function

```
Print Bin$(3)                                    ' Prints "11"

' Converts Double argument to Long.
Print Bin$(3.0)                                  ' Prints "11"

' Rounds Double argument, then converts to Long.
Print Bin$(3.3)                                  ' Prints "11"

' Computes product 2.79, rounds to 3.0, then converts to Long.
Print Bin$(3.1 * .9)                             ' Prints "11"
```

Bracket notation

For applications built in some Lotus products, such as 1-2-3®, you can use names in brackets rather than object reference variables to identify Lotus product objects. To determine whether your Lotus product supports this notation, see the product documentation.

Syntax

[*prodObjName***]**

Elements

prodObjName

The name understood by the product to identify an object (an instance of a product class).

Usage

In some cases, Lotus products assign names to objects, and in other cases you can use the product user interface to name the objects you create. In a spreadsheet, for example, A1 identifies a particular cell, and you could use the user interface to name a chart SalesTracking.

Bracket notation lets you use these names without declaring an object variable and binding it to the object. For example, the product might allow you to use:

```
[A1].Value = 247000
```

instead of:

```
Dim myCell As Cell
Set myCell = Bind Cell("A1")
myCell.Value = 247000
```

In some cases, the product uses bracket notation when it records transcripts of user actions. This makes the transcripts easier to read and modify. For more information, see the product documentation.

The LotusScript compiler does not attempt to determine the class of objects that are identified with bracket notation, so any class syntax errors you make (such as the incorrect use of properties and other methods), will generate run-time errors, not compile-time errors.

You can also use empty brackets to identify the currently selected product object. Empty brackets are equivalent to leading dot notation. For example, if the current selection is a range named SalesQuotas, then

```
[].Print
```

and

```
.Print
```

are equivalent to

```
[SalesQuotas].Print
```

All three statements print the contents of the SalesQuotas range.

To include square brackets as text within a string, double the brackets. For example, if the current selection is a range named SalesQuotas[East], use the following syntax:

```
[SalesQuotas[[East]]].Print
```

Examples: Bracket notation

```
' Use the Chart class Print method to print the chart SalesTracking.
[SalesTracking].Print
```

Call statement

Calls a LotusScript sub or function.

Syntax 1

Call *subOrFunction* [**(** [*argList*] **)**]

Syntax 2

subOrFunction [*argList*]

Syntax 3

subOrFunction **(** *argPassedByVal* **)**

Syntax 4 (functions only)

returnVal = *function* [**(** [*argList*] **)**]

Elements

subOrFunction
The name of the sub or function being called.

argList
A comma-separated argument list for the sub or function being called.

argPassedByVal

A single argument to be passed by value to the sub or function being called.

function
The name of the function being called.

returnVal
The assignment variable containing the function's return value.

Usage

When you use the Call keyword, you must include parentheses around the argument list. If there are no arguments, the empty parentheses are optional.

When you omit the Call keyword, the following parenthesis rules apply:

- For a sub or a function, do not use parentheses around the argument list (Syntax 2) unless you are passing a single argument by value to the sub or function (Syntax 3).
- For a function within an expression, enclose the argument list (if there is one) in parentheses (Syntax 4).

Sub calls do not return a value.

LotusScript uses a function's return value if the function call appears in an expression. The call can appear anywhere in an expression where the data type of the function's return value is legal. Function calls that use the Call keyword, however, do not return a value and cannot appear in an expression.

LotusScript always uses the return value of a call to a built-in function. You must use its return value in an expression, and you cannot use the Call keyword.

Referencing a function that returns an array, list, or collection

If a function returns an array, list, or collection, a reference to the function can contain subscripts according to the following rules:

- If the function has parameters, the first parenthesized list following the reference must be the argument list. A second parenthesized list is treated as a subscript list. For example, f1(1,2)(3) is a reference to a function f1 that has two parameters and returns a container.
- If the function has no parameters and the return type is a variant or collection object, two parenthesized lists, but not one, can follow the reference. The first must be empty and the second is treated as a subscript list. For example, f1()(3) is a reference to a function f1 that contains no parameters but is a container.
- If the function has no parameters and the return type is not a variant or collection object, any parenthesized list following the reference is an error, except that a single empty list is allowed. For example, f1() is a reference to a function f1 that contains no parameters and may or may not be a container; if f1 is a container, the reference is to the entire container.

Examples: Call statement

Example 1

```
' Define a function and then invoke it in three ways.

Function MiniMult (x As Integer, y As Integer) As Integer
   MiniMult = x% * y%
End Function
Dim result As Integer

Call MiniMult(3, 4)         ' With Call; return value (12) is not
used.
MiniMult 3, 4               ' Without Call; return value is not used.
result% = MiniMult(3, 4)    ' With Call; return value is used.
Print result                ' Prints 12.
```

Example 2

```
' Define a sub and then invoke it in two ways.

Sub PrintProduct(a As Integer, b As Integer)
   Print a% * b%
End Sub

Call PrintProduct(34, 5)    ' With Call; prints 170.
PrintProduct 34, 5          ' Without Call; prints 170.
```

CCur function

Returns a value converted to the Currency data type.

Syntax

CCur (*expr* **)**

Elements

expr

Any numeric expression, or a string expression that LotusScript can convert to a number.

Return value

CCur returns the numeric value of *expr* rounded to four decimal places, as a Currency value.

CCur(EMPTY) returns 0.

If *expr* is a string expression, CCur returns the numeric representation of the string, rounded to four decimal places. If LotusScript cannot convert the string to a number, the function returns an error.

If the value of *expr* is too large to fit in the Currency data type, the function returns an error.

Examples: CCur function

```
Dim bulkPrice As Double
Dim labelPrice As String
Dim unitsSold As Integer
Dim paymentDue As Currency

bulkPrice# = 11.400556
unitsSold% = 57
paymentDue@ = CCur(bulkPrice# * unitsSold%)
Print paymentDue@                        ' Prints 649.8317

labelPrice$ = "12.99"
paymentDue@ = CCur(labelPrice$) * unitsSold%
Print paymentDue@                        ' Prints 740.43
```

CDat function

Converts a numeric value or string value to a date/time value.

Syntax

CDat (*expr* **)**

CVDate is acceptable in place of CDat.

Elements

expr

Any of the following kinds of expression:

- A numeric expression
- A string expression that can be converted to a number
- A string expression that can be converted to a date/time value

Return value

CDat returns a date/time value.

The data type of the return value is a Variant of DataType 7 (Date/Time).

If the integer part of *expr* is not in the range -657434 to 2958465, the function returns an error.

CDat(0) returns the date December 30, 1899, formatted as 12/30/1899. CDat(EMPTY) returns the same value.

Usage

CDat converts *expr* to a date/time value in the LotusScript date/time format.

CDat uses different conversion rules depending on the form of *expr*:

- If *expr* is a numeric expression, CDat converts the integer part of its value to a date and the fractional part to a time, and returns the corresponding date/time value.

  A date/time value stored in a Variant is an eight-byte floating-point value. The integer part represents a serial day counted from Jan 1, 100 AD. Valid dates are represented by integer numbers in the range -657434, representing Jan 1, 100 AD, to 2958465, representing Dec 31, 9999 AD. The fractional part represents the time as a fraction of a day, measured from time 00:00:00 (midnight on the previous day). In this representation of date/time values, day 1 is the date December 31, 1899.

- If *expr* is a string expression that can be converted to a number, CDat converts the string to a number and then converts the number to a date/time value and returns the result, as described above.

- If *expr* is a string expression in the form of a date, for example "4/20/95", CDat converts the value to a date/time in the internal date/time format.

If LotusScript cannot convert the value to a date/time, the function returns an error.

Examples: CDat function

```
Dim dateV As Variant

' Convert a numeric value to a date/time value.
dateV = CDat(34814.3289)

' Display the formatted date and time.
Print Format$(dateV, "Medium Date"), _
      Format$(dateV, "Medium Time")
' Prints 25-Apr-95  07:53 AM

' Convert the date back to a number.
Print CDbl(dateV)                       ' Prints 34814.3289

' Convert a date string to a date.
Print CDat("April 25, 1995")            ' Prints 4/25/95
```

CDbl function

Returns a value converted to the Double data type.

Syntax
CDbl (*expr* **)**

Elements
expr

Any numeric expression, or a string expression that LotusScript can convert to a number.

Return value
CDbl returns the numeric value of *expr* as a Double value.

CDbl(EMPTY) returns 0.

If *expr* is a string expression, CDbl returns the numeric representation of the string, including any fractional part. If LotusScript cannot convert the string to a number, the function returns an error.

If the value of *expr* is too large to fit in the Double data type, the function returns an error.

Examples: CDbl function
```
' Convert the sum of two Single values to Double.
Dim x As Single
Dim y As Single
Dim result As Double
x! = 11.06E23
y! = 6.02E23
result# = CDbl(x! + y!)
Print result#                          ' Prints 1.70800003057064E+24
```

ChDir statement

Sets the current directory.

Syntax
ChDir *path*

Elements
path

A string expression representing the path of an existing directory.

Usage

ChDir sets the current directory to *path*. The current directory is the directory that LotusScript uses when you specify a file name without a path.

If the value of *path* does not begin with a drive letter, ChDir sets the current directory for the current drive.

If the value of *path* includes a drive letter, ChDir sets the current directory for that drive, but does not reset the current drive. The path will not be used as the current directory until the current drive is reset. To change the current drive, use ChDrive.

To return the current drive, use CurDrive. To return the current directory, use CurDir.

The format and maximum length of *path* follow the conventions of the platform on which LotusScript is running.

Examples: ChDir statement

```
' Set the current drive to d.
ChDrive "d"

' Set current directory on the c drive to \test.
ChDir "c:\test"

' Set current directory on current drive (d) to \test.
ChDir "\test"

Print CurDir()                          ' Prints d:\test
```

ChDrive statement

Sets the current drive.

Syntax

ChDrive *drive*

Elements

drive

A string expression representing an existing drive.

Usage

ChDrive sets the current drive to the value of *drive*. The current drive is the drive that LotusScript uses whenever you specify a file name or a path that does not include a drive.

If the value of *drive* is the empty string (""), ChDrive does not change the current drive.

If the value of *drive* is a string of more than one character, ChDrive uses only the first character. ChDrive does not require a colon (:) after the drive letter.

The *drive* must be in the range A to *lastdrive*, inclusive, where *lastdrive* is the maximum drive letter specified in CONFIG.SYS.

To change the current directory, use ChDir.

To return the current drive, use CurDrive. To return the current directory, use CurDir.

Examples: ChDrive statement

```
' Set the current drive to D.
ChDrive "D"
```

Chr function

Returns the character represented by a platform-specific numeric character code.

Syntax

Chr[**$**] **(** *numExpr* **)**

Elements

numExpr

A numeric expression of data type Long, representing a numeric character code in the platform character-code set. Its legal range is the range of the platform character-code set.

Return value

Chr returns the platform-specific character corresponding to the value of *numExpr*.

Chr returns a Variant of DataType 8 (String). Chr$ returns a String.

Usage

If the value of *numExpr* contains a fraction, LotusScript rounds the value before using it.

Examples: Chr function

```
Dim myAlph As String
Dim letterCode As Long
' Iterate through the character codes for "a" through "z".
' Build an alphabet string by concatenating the letters.
For letterCode& = Asc("a") To Asc("z")
   myAlph$ = myAlph$ & Chr$(letterCode&)
Next
Print myAlph$                  ' Prints abcdefghijklmnopqrstuvwxyz
```

CInt function

Returns a value converted to the Integer data type.

Syntax

CInt (*expr* **)**

Elements

expr

Any numeric expression, or a string expression that LotusScript can convert to a number.

Return value

CInt returns the value of *expr* rounded to the nearest integer, as an Integer value.

CInt(EMPTY) returns 0.

If *expr* is a string expression, CInt returns the numeric representation of the string, rounded to the nearest integer. If LotusScript cannot convert the string to a number, the function returns an error.

If the value of *expr* is too large to fit in the Integer data type, the function returns an error.

Examples: CInt function

```
' Convert a Currency value to Integer.
Dim x As Currency
x@ = 13.43
Print CInt(x@)                  ' Prints 13
```

Class statement

Defines a class with its member variables and procedures.

Syntax

[**Public** | **Private**] **Class** *className* [**As** *baseClass*]

classBody

End Class

Elements

Public | **Private**

Optional. Public specifies that the class is visible outside the module where the class is defined, as long as this module is loaded. Private specifies that the class is visible only in this module.

A class is Private by default.

className

The name of the class.

baseClass

Optional. The name of another class from which this class is derived.

classBody

Declarations and definitions of class members. Class members can include member variables; member procedures (functions, subs, and properties); a constructor sub, named New; and a destructor sub, named Delete. Constants cannot be class members.

Usage

The Public keyword cannot be used in a product object script or %Include file in a product object script, except to declare class members. You must put such Public declarations in (Globals).

Rules for defining classes:

- Define a class only in module scope. Do not define a class within a procedure or within another class.
- Do not use the word Object as a class name.

Rules for declaring member variables:

- Omit the Dim keyword from the variable declaration of member variables.
- A separate declaration is required for each member variable. You can't declare two or more member variables in a single declaration using a comma-separated list.
- You can use the Public or Private keywords for variable declarations. A member variable is private by default; it can be accessed only within the class.
- Member variables cannot be declared Static.
- A class can include an instance of itself as a member, but the variable declaration cannot include the New keyword. That is, the variable declaration cannot create an object of the class.
- Do not use the following LotusScript keywords as member variable names: Public, Private, Static, Sub, Function, Property, Get, Set, New, Delete, and Rem.

Rules for declaring member procedures:

- You can use the keywords Public or Private for procedure declarations. A member procedure is Public by default; it can be accessed outside of the class.
- Member procedures cannot be declared Static.
- All LotusScript keywords are legal as member procedure names. Use the names New and Delete only to name the class constructor and destructor subs, respectively.

Rules for referring to class members:

- Refer to class members using the notation *objName.memberName*, where *memberName* identifies a class member defined in the class of the object reference variable *objName*.
- You can use the keyword Me to refer to the object itself when you are inside a member procedure. In the example, Me.textColor refers to the value currently assigned to the textColor member of this instance of the class.
- If you name a class member with a LotusScript keyword, you must refer to the member within member subprograms using the Me keyword.
- Derived class methods can override methods of the base class. The signature of the overriding member must match the signature of the overridden member. Within a derived class's procedure, you refer to a base class member of the same name using the notation *baseClassName..memberName*.
- Use the With statement to work with members of a specific class using the notation *.memberName*.

Rules for working with objects (class instances):

- To create an object, use the New keyword in a Dim or Set statement for an object reference variable.
- LotusScript sets the initial value of an object reference variable to NOTHING. Use the Is operator to test an object reference variable for the NOTHING value.
- Any Variant variable can take an object reference as its value. Use the IsObject function to test whether the contents of a Variant variable is an object reference.
- Use the Delete statement to delete an object. LotusScript sets the value of variables that refer to the object to NOTHING.

A class definition can include a definition for the constructor sub, named New. If the definition exists, LotusScript calls this sub each time it creates an object of that class.

A class definition can include a definition for the destructor sub, named Delete. If the definition exists, LotusScript calls this sub whenever it deletes an object of that class.

Examples: Class statement

```
' Define a class.
Class textObject

   ' Declare member variables.
   backGroundColor As Integer
   textColor As Integer
   contentString As String

   ' Define constructor sub.
   Sub New (bColor As Integer, tColor As Integer, _
      cString As String)
      backGroundColor% = bColor%
      textColor% = tColor%
      contentString$ = cString$
   End Sub

   ' Define destructor sub.
   Sub Delete
      Print "Deleting text object."
   End Sub

   ' Define a sub to invert background and text colors.
   Sub InvertColors
      Dim x% As Integer, y% As Integer
      x% = backGroundColor%
      y% = textColor%
      Me.backGroundColor% = y%
      Me.textColor% = x%
   End Sub

End Class

' Create a new object of class textObject.
Dim y As textObject
Set y = New textObject(0, 255, "This is my text")

' Invert the object's background and text colors.
y.InvertColors

' Delete the object.
Delete y
' Output:
' Deleting text object.
```

CLng function

Returns a value converted to the Long data type.

Syntax

CLng (*expr* **)**

Elements

expr

Any numeric expression, or a string expression that LotusScript can convert to a number.

Return value

CLng returns the value of *expr* rounded to the nearest integer, as a Long value.

CLng(EMPTY) returns 0.

If *expr* is a string expression, CLng returns the numeric representation of the string, rounded to the nearest integer. If LotusScript cannot convert the string to a number, the function returns an error.

If the value of *expr* is too large to fit in the Long data type, the function returns an error.

Examples: CLng function

```
' Convert a Double value to Long.
Dim x As Double
x# = 13.400556
Print CLng(x#)                  ' Prints 13
```

Close statement

Closes one or more open files, after writing all internally buffered data to the files.

Syntax

Close [[**#**] *fileNumber* [**,** [**#**] *fileNumber*] ...]

Elements

fileNumber

Optional. The number that LotusScript assigned to the file when it was opened.

If you omit *fileNumber,* Close closes all open files.

Usage

The pound sign (#) preceding *fileNumber* is optional and has no effect on the statement.

Before closing the open files, Close writes all internally buffered data to the files.

If LotusScript encounters a run-time error that is not handled by an On Error statement, LotusScript closes all open files; otherwise, the files remain open.

If the value of *fileNumber* contains a fraction, LotusScript rounds the value before using it.

Examples: Close statement

```
Open "c:\rab.asc" For Input Access Read Shared As 1 Len = 128
Close #1
```

Command function

Returns the command-line arguments used to start the Lotus product that started LotusScript.

Syntax

Command[**$**]

Return value

The return value does not include the program name.

Command returns a Variant of DataType 8 (String). Command$ returns a String.

If the command that started the product specified no arguments, the function returns the empty string ("").

Usage

You can call the Command function as either Command or Command(). You can call the Command$ function as either Command$ or Command$().

To run a Lotus product macro in a script, use Evaluate. To start a program from a script, use Shell.

Examples: Command function

```
If Command$() = "" Then
   Print "No command-line arguments"
Else
   Print "Command-line arguments are: " + Command$()
End If
```

Const statement

Defines a constant.

Syntax

[**Public** | **Private**] **Const** *constName* = *expr* [**,** *constName* = *expr*]...

Elements

Public | **Private**

Optional. Public specifies that the constant is visible outside the module where the constant is defined, as long as that module is loaded. Private specifies that the constant is visible only within the module where the constant is defined.

A constant is Private by default.

If you declare a constant within a procedure, you cannot use Public or Private.

constName

The name of the constant.

expr

An expression. The value of the expression is the value of the constant.

The expression can contain any of the following.

- Literal values (numbers and strings)
- Other constants
- Arithmetic and logical operators
- Built-in functions, if their arguments are constant and if LotusScript can evaluate them at compile time. The following functions are evaluated at compile time if their arguments are expressions including only literals and constants.

Functions that can be evaluated as LotusScript constants

Abs	InStrB	RightB
ACos	Int	Round
ASin	LCase	RTrim
ATn	Left	Sgn
ATn2	LeftB	Sin
Bin	Len	Sqr
Cos	LenB	Str
DataType	LenBP	Tan
DateNumber	Log	TimeNumber
Exp	LTrim	Trim
Fix	Mid	TypeName
Fraction	MidB	UCase
Hex	Oct	Val
InStr	Right	

Usage

The Public keyword cannot be used in a product object script or %Include file in a product object script, except to declare class members. You must put such Public declarations in (Globals).

A constant is a named variable whose value cannot be changed. You can declare a constant in a module or a procedure, but you cannot declare a constant in a type or class definition.

You can specify the data type of a constant by appending a data type suffix character to *constName*. Alternatively, if the constant is numeric and *expr* is a numeric literal, you can specify the data type by appending a data type suffix character to *expr*.

If you do not append a data type suffix character to *constName* or *expr*, LotusScript determines the data type of the constant by the value assigned to it.

- For a floating-point value, the data type is Double.
- For an integer value, the data type is Integer or Long, depending on the magnitude of the value.

These rules are illustrated in the examples following.

Whether you specify a suffix character in the Const statement or LotusScript determines the data type based on the constant's value, you can use the constant in a script with or without a data type suffix character. If you use the constant with a suffix character, the suffix character must match the data type of the constant.

The data type of a constant is not affected by Def*type* statements.

Examples: Const statement

Example 1

```
Const x = 123.45    ' Define a Double constant.
Const y = 123       ' Define an Integer constant.
Const z = 123456    ' Define a Long constant. The value is too large
                    ' to define an Integer constant.
```

Example 2

```
' Define a String constant, firstName.
Const firstName$ = "Andrea"
' Define a Single constant, appInterest.
Const appInterest! = 0.125
' Define a Currency constant, appLoan.
Const appLoan@ = 4350.20

' Display a message about the amount of interest owed.
MessageBox firstName$ & " owes "  _
   & Format(appLoan@ * appInterest!, "Currency")
```

Cos function

Returns the cosine of an angle.

Syntax
Cos (*angle* **)**

Elements
angle

A numeric expression, specifying an angle expressed in radians.

Return value
Cos returns the cosine of *angle,* a value between -1 and 1, inclusive.

The data type of the return value is Double.

Examples: Cos function
```
Dim degrees As Integer
Dim rad As Double

' Convert the angle 45 degrees to radians.
degrees% = 45
rad# = degrees% * (PI / 180)

' Print the cosine of that angle.
Print Cos(rad#)                ' Prints .707106781186548
```

CreateObject function

Creates an OLE Automation object of the specified class.

Note CreateObject is not supported under OS/2, under UNIX, or on the Macintosh.

Syntax
CreateObject (*className* **)**

Elements
className

A string of the form *appName.appClass,* designating the kind of object to create (for example, “WordPro.Application”).

The *appName* is an application that supports OLE Automation.

The *appClass* is the class of the object to create. Products that support OLE Automation provide one or more classes. See the product documentation for details.

Return value
CreateObject returns a reference to an OLE Automation object.

Usage
Use the Set statement to assign the object reference returned by CreateObject to a Variant variable.

If the application is not already running, CreateObject starts it before creating the OLE Automation object. References to the object remain valid only while the application is running. If the application terminates while you are using the object reference, LotusScript returns a run-time error.

LotusScript supports the OLE vartypes listed in the table below. Only an OLE method or property can return a vartype designated as "OLE only."

OLE vartype	*Description*
VT_EMPTY	(No data)
VT_NULL	(No data)
VT_I2	2-byte signed integer
VT_I4	4-byte signed integer
VT_R4	4-byte real
VT_R8	8-byte real
VT_CY	Currency
VT_DATE	Date
VT_BSTR	String
VT_DISPATCH	IDispatch, OLE only
VT_ERROR	Error, OLE only
VT_BOOL	Boolean
VT_VARIANT	(A reference to data of any other type)
VT_UNKNOWN	IUnknown, OLE only
VT_ARRAY	(An array of data of any other type)

LotusScript supports iterating over OLE collections with a ForAll statement.

LotusScript supports passing arguments to OLE properties. For example:

```
' Set v.prop to 4; v.prop takes two arguments.
v.prop(arg1, arg2) = 4
```

LotusScript does not support identifying arguments for OLE methods or properties by name rather than by the order in which they appear, nor does LotusScript support using an OLE name by itself (without an explicit property) to identify a default property.

Results are unspecified for arguments to OLE methods and properties of type boolean, byte, and date that are passed by reference. LotusScript does not support these data types.

The word CreateObject is not a LotusScript keyword.

Examples: CreateObject function

This example creates a Notes session and displays some information from it.

```
' Create a Notes session and display the name of the current user.
Dim session As Variant
Set session = CreateObject("Notes.NotesSession")
Messagebox session.UserName
```

CSng function

Returns a value converted to the Single data type.

Syntax

CSng (*expr* **)**

Elements

expr

Any numeric expression, or a string expression that LotusScript can convert to a number.

Return value

CSng returns the numeric value of *expr* as a Single value.

CSng(EMPTY) returns 0.

If *expr* is a string expression, CSng returns the numeric representation of the string, including any fractional part. If LotusScript cannot convert the string to a number, the function returns an error.

If the value of *expr* is too large to fit in the Single data type, the function returns an error.

Examples: CSng function

```
' Convert a Double value by rounding to nearest Single.
Dim x As Double
x# = 1.70800003057064E+24
Print CSng(x#)              ' Prints 1.708E+24
```

CStr function

Returns a value converted to the String data type.

Syntax

CStr (*expr* **)**

expr

Any numeric expression, or a string expression that LotusScript can convert to a number.

Return value

CStr returns the value of *expr* as a String value.

CStr(EMPTY) returns the empty string ("").

Examples: CStr function

```
Dim x As Integer
Dim y As Integer
x% = 1
y% = 2

' Use the addition operator +
Print x% + y%                          ' Prints 3

' Use the string concatenation operator +
Print CStr(x%) + CStr(y%)              ' Prints 12
```

CurDir function

Returns the current directory on a specified drive.

Syntax

CurDir[**$**] [**(** *drive* **)**]

Elements

drive

Optional. A string expression specifying an existing drive. If you omit *drive*, CurDir uses the current drive.

Return value

CurDir returns the current directory on *drive.*

CurDir returns a Variant of DataType 8 (String). CurDir$ returns a String.

Usage

If the value of *drive* is a string of more than one character, CurDir uses only the first character. CurDir does not require a colon after the drive letter.

To set the current directory on a specified drive, use ChDir. To set the current drive, use ChDrive. To return the current drive, use CurDrive.

You can call this function with no arguments as either CurDir or CurDir( ).

Examples: CurDir function

```
ChDir "c:\test"
Print CurDir$()   ' Prints "c:\test"
```

CurDrive function

Returns a string identifying the current drive.

Syntax

CurDrive[**$**]

Return value

CurDrive returns the current drive letter followed by a colon.

CurDrive returns a Variant of DataType 8 (String). CurDrive$ return a String.

To set the current directory on a specified drive, use ChDir. To set the current drive, use ChDrive. To return the current directory on a drive, use CurDir.

You can call the CurDrive function as either CurDrive or CurDrive(). You can call the CurDrive$ function as either CurDrive$ or CurDrive$().

Examples: CurDrive function

```
Dim tempDrive As String
tempDrive$ = CurDrive$()
If tempDrive$ <> "c:" Then
   ChDrive "c"
End If
ChDir "\test"
Print CurDir$()   ' Prints "c:\test"
```

Currency data type

Specifies a variable that contains an 8-byte integer, scaled to four decimal places to suitably represent a monetary value.

Usage

The Currency suffix character for implicit type declaration is @.

Use the Currency data type for calculations with money.

Currency variables are initialized to 0.

Currency values must fall within the range -922,337,203,685,477.5807 to 922,337,203,685,477.5807 inclusive.

Use the Currency data type for fixed point calculations in which four-decimal-place accuracy is meaningful.

LotusScript aligns Currency data on a 4-byte boundary. In user-defined types, declaring variables in order from highest to lowest alignment boundaries makes the most efficient use of data storage space.

Examples: Currency data type

```
' Explicitly declare two Currency variables.
Dim sales As Currency
Dim expenses As Currency
sales@ = 20.9999
expenses@ = 10.5555
' Implicitly declare a Currency variable.
earnings@ = sales@ - expenses@
' Currency is calculated to four decimal places.
Print earnings@                        ' Prints 10.4444
```

CVar function

Returns a value converted to the Variant data type.

Syntax

CVar (*expr* **)**

Elements

expr

Any expression.

Return value

CVar returns the value of *expr*.

The data type of the return value is Variant.

Examples: CVar function

```
' The Abs function requires a numeric or Variant argument.
' Convert a string value to Variant and use it in Abs.
Dim gNum As String
gNum$ = "-1"
Print Abs(CVar(gNum$))          ' Prints 1 (absolute value of -1)
Print Abs(gNum$)                ' Generates an error
```

DataType function

Returns the data type of the value of an expression.

Syntax

DataType (*expr* **)**

VarType is acceptable in place of DataType.

Elements

expr

Any expression.

Return value

DataType returns a number representing the data type of *expr*.

The following table describes the possible return values. The first column is the return value. The last column is "Yes" if the return value applies to variants only.

Return	*Value type*	*Constant*	*Variant*
0	EMPTY	V_EMPTY	Yes
1	NULL	V_NULL	Yes
2	Integer	V_INTEGER	
3	Long	V_LONG	
4	Single	V_SINGLE	
5	Double	V_DOUBLE	
6	Currency	V_CURRENCY	
7	Date/Time	V_DATE	Yes
8	String	V_STRING	
9	OLE object or NOTHING	V_DISPATCH	Yes
10	OLE error	V_ERROR	Yes
11	Boolean	V_BOOLEAN	Yes
12	Variant list or array	V_VARIANT	

continued

Return	*Value type*	*Constant*	*Variant*
13	IUNKNOWN (OLE value)	V_IUNKNOWN	Yes
34	User-defined object	V_LSOBJ	
35	Product object	V_PRODOBJ	
2048	List		
8192	Fixed array		
8704	Dynamic array		

Usage

The file lsconst.lss defines the constants described in the preceding table. If you want to refer to the return values as symbolic constants instead of numbers, use the %Include directive to include this file in your script.

If the argument to DataType is a list or an array, the return value is the sum of the value that represents a list or an array plus the value that represents the data type of elements of the list or array. For example, a fixed array of Integers is 8194 (that is, 8192 + 2); a list of Variants is 2060 (that is, 2048 + 12).

The return value 13 signifies an unknown value type, corresponding to the OLE value IUNKNOWN. To test for this value, use the IsUnknown function.

Examples: DataType function

```
Dim item(5) As Variant          ' Declare a Variant fixed array.
Dim itemWeight As Single
Dim itemName As String

itemWeight! = 2.7182
itemName$ = "Jute twine"

item(1) = itemWeight!
item(2) = itemName$
Print DataType(item(1))         ' Prints 4
Print DataType(item(2))         ' Prints 8
Print DataType(item(3))         ' Prints 0 (initalized to EMPTY)

Dim cells As Range              ' Suppose Range is a
                                ' product-defined class.
Print DataType(cells)           ' Prints 35
Set cells2 = cells
Print DataType(cells2)          ' Prints 35
Dim areas(3) As Range           ' An array of Range product objects
Print DataType(areas)           ' Prints 8227 (8192 + 35)

Set cal = CreateObject("dispcalc.ccalc")
Print DataType(cal)             ' Prints 9
```

```
Dim stats(3) As Integer          ' An array of Integers
Print DataType(stats%)           ' Prints 8194 (8192 + 2)

Dim misc List As Variant         ' A list of Variants
Print DataType(misc)             ' Prints 2060 (2048 + 12)
```

Data types

LotusScript recognizes the following scalar (numeric and string) data types:

Data type	*Suffix*	*Value range*	*Size*
Integer	%	-32,768 to 32,767 Initial value: 0	2 bytes
Long	&	-2,147,483,648 to 2,147,483,647 Initial value: 0	4 bytes
Single	!	-3.402823E+38 to 3.402823E+38 Initial value: 0	4 bytes
Double	#	-1.7976931348623158+308 to 1.7976931348623158+308 Initial value: 0	8 bytes
Currency	@	-922,337,203,685,477.5807 to 922,337,203,685,477.5807 Initial value: 0	8 bytes
String	$	(String length ranges from 0 to 32K (2 bytes / character) characters) Initial value: “” (empty string)	

Besides these scalar data types, LotusScript supports the following additional data types and data structures:

Data type or structure	*Description*	*Size*
Array	An aggregate set of elements having the same data type. An array can comprise up to 8 dimensions whose subscript bounds can range from -32768 to 32767. Initial value: Each element in a fixed array has an initial value appropriate to its data type.	Up to 64K bytes
List	A one-dimensional aggregate set whose elements have the same data type and are referred to by name rather than by subscript.	Up to 64K bytes

continued

Data type or structure	*Description*	*Size*
Variant	A special data type that can contain any scalar value, array, list, or object reference. Initial value: EMPTY	16 bytes
User-defined data type	An aggregate set of elements of possibly disparate data types. Comparable to a record in Pascal or a struct in C. Initial value: Member variables have initial values appropriate to their data types.	Up to 64K bytes
Class	An aggregate set of elements of possibly disparate data types together with procedures that operate on them. Initial value: When you create an instance of a class, LotusScript initializes its member variables to values appropriate to their data types, and generates an object reference to it.	
Object reference	A pointer to an OLE Automation object or an instance of a product class or user-defined class. Initial value: NOTHING.	4 bytes

In each of the preceding tables, the specified storage size is platform-independent.

Date function

Returns the current system date as a date/time value.

Syntax

Date[**$**]

Return value

Date returns the integer part of the value returned by the Now function. Date returns that value as a Variant of DataType 7 (Date/Time). Date$ returns that value as a String.

Usage

The Date function is equivalent to the Today function.

You can call the Date function as either Date or Date( ). You can call the Date$ function as either Date$ or Date$()

Examples: Date function

```
Print Date$         ' Prints "04/25/95" if the current
                    ' system date is April 25, 1995.
```

Date statement

Sets the system date.

Syntax

Date[**$**] **=** *dateExpr*

Elements

dateExpr

Any expression whose value is a valid date/time value: either a String in a valid date/time format, or else a Variant containing either a date/time value or a string value in date/time format.

If *dateExpr* is a string in which the date part contains only numbers and valid date separators, the operating system's international Short Date format determines the order in which the numbers are interpreted as month, day, and year values. The date part of the string must have one of the following forms:

mm-dd-yy or dd-mm-yy
mm-dd-yyyy or dd-mm-yyyy
mm/dd/yy or dd/mm/yy
mm/dd/yyyy or dd/mm/yyyy

Usage

For a string date/time value, LotusScript interprets a 2-digit year designation as a year in the twentieth century. For example, "1/1/17" and "1/1/1917" are considered the same.

Examples: Date statement

```
' Depending on the international Short Date format,
' set the system date to September 7, 1995 or to 9 July, 1995.
Date$ = "09-07-95"
```

DateNumber function

Returns a date value for a given set of year, month, and day numbers.

Syntax

DateNumber (*year* **,** *month* **,** *day* **)**

DateSerial is acceptable in place of DateNumber.

Elements

year

A numeric expression designating a year. 0 through 99 designate the years 1900 through 1999. To specify another century, use all four digits: for example, 1895.

month

A numeric expression designating a month of the year (a value from 1 through 12).

If you assign *month* a negative value, DateNumber calculates a prior date by measuring backward from December of the preceding year. (For example, 1995, -2, 10 is evaluated as October 10, 1994.)

day

A numeric expression designating a day of the month (a value from 1 through 31).

If you assign *day* a negative value, then DateNumber calculates a prior date by measuring backward from the last day of the month immediately preceding the specified month. (For example, 1995, 5, -3 is evaluated as April 27, 1995, by subtracting 3 from 30, the last day of April, the month before the 5th month.)

Return value

DateNumber returns the date value for the given *year*, *month*, and *day*.

The data type of the return value is a Variant of DateType 7 (Date/Time).

Examples: DateNumber function

```
Print DateNumber(1947, 10, 8)              ' Prints 10/8/47

' The following two functions calculate a past date
' using negative arguments.
' Print the date 5 months and 10 days before 2/4/95.
Print DateNumber(95, 2 - 5, 4 - 10)        ' Prints 8/25/94
' Print the date 3 months and 6 days before 1/1/95.
Print DateNumber(95, -3, -6)               ' Prints 8/25/94
```

DateValue function

Returns the date value represented by a string expression.

Syntax

DateValue (*stringExpr* **)**

Elements

stringExpr

A string expression representing a date/time. *stringExpr* must be a String in a valid date/time format or else a Variant containing either a date/time value or a string value in date/time format. If you omit the year in *stringExpr*, DateValue uses the year in the current system date.

If *stringExpr* is a string whose date part contains only numbers and valid date separators, the operating system's international Short Date format determines the order in which the numbers are interpreted as month, day, and year values.

In a string representation of a date/time, a 2-digit designation of a year is interpreted as that year in the twentieth century. For example, 17 and 1917 are equivalent year designations.

Return value

DateValue returns the date value represented by *stringExpr*.

The data type of the return value is a Variant of DataType 7 (Date/Time).

Usage

If the *stringExpr* argument specifies a time of day, DateValue validates the time, but omits it from the return value.

Examples: DateValue function

```
Dim birthDateV As Variant
' Calculate the date value for October 8, 1947.
birthDateV = DateValue("October 8, 1947")
' Print this value as a date string.
Print CDat(birthDateV)   ' Prints 10/8/47
' Print the age this person reaches, in years,
' on this year's birthday.
Print Year(Today) - Year(birthDateV)
```

Day function

Returns the day of the month (an integer from 1 to 31) for a date/time argument.

Syntax

Day (*dateExpr* **)**

Elements

dateExpr

Any of the following kinds of expression:

- A valid date/time string of String or Variant data type. In a date/time string, a 2-digit designation of a year is interpreted as that year in the twentieth century. For example, 17 and 1917 are equivalent year designations.
- A numeric expression whose value is a Variant of DataType 7 (Date/Time)
- A number within the valid date range: the range -657434 (representing Jan 1, 100 AD) to 2958465 (Dec 31, 9999 AD)
- NULL

Return value

Day returns an integer between 1 and 31.

The data type of the return value is a Variant of DataType 2 (Integer).

Day(NULL) returns NULL.

Examples: Day function

```
Dim x As Variant, dd As Integer
x = DateNumber(1992, 4, 7)
dd% = Day(x)
Print dd%            ' Prints 7
```

Declare statement (external C calls)

Declares a LotusScript function or sub that calls an external C function, allowing calls to a function that is defined in a shared library of C routines.

Syntax

Declare [**Public** | **Private**] { **Function** | **Sub** } *LSname* **Lib** *libName* [**Alias** *aliasName*] **(** [*argList*] **)** [**As** *returnType*]

Elements

Public | **Private**

Optional. Public indicates that the declared C function is visible outside this module, for as long as the module is loaded. Private indicates that the declared C function is visible only within this module.

A declared C function is Private by default.

Function | **Sub**

Specifies that the C function is to be called as either a function or a sub.

LSname

The function or sub name used within LotusScript. If you omit the Alias clause, this name must match the name declared in the shared library.

If the statement is declaring a Function (using the keyword Function), then you can append a data type suffix character to *LSname*, to declare the type of the function's return value.

libName

A literal string, or a string constant, specifying the shared library file name. The file name extension is optional. You can optionally include a complete path specification. LotusScript automatically converts *libName* to uppercase. If you need to preserve case sensitivity, use the *aliasName* described below.

aliasName

Optional. A literal string containing one of the following:

- A case-sensitive C function name as declared in the shared library.
- A pound sign (#) followed by an ordinal number representing the position of the function in the library; for example, "#1".

This argument is useful when the C function name is not a valid LotusScript name, or when you need to preserve case sensitivity (for example, when calling an exported library function in a 32-bit version of Windows).

argList

Optional. An argument list for the external function. Parentheses enclosing the list are required, even if the C function takes no arguments.

argList has the form:

argument [**,** *argument*] ...

where *argument* is:

[**ByVal**] *name* **As** [**LMBCS** | **Unicode**] [*dataType* **| Any**]

The optional LMBCS and Unicode keywords may be used with the String data type only, to specify the character set. See the usage information and examples that follow.

Use the keyword Any to pass an argument to a C function without specifying a data type, suppressing type checking.

returnType

The data type of the function's return value. The clause **As** *returnType* is not allowed for a sub, since a sub doesn't return a value.

For a function, either specify **As** *returnType*, or append a data type suffix character to *LSname,* to declare the data type of the function's return value. Do not specify both a *returnType* and a data type suffix character.

You can't use Any as a *returnType.*

You can't use Variant, Currency, or fixed-length String as a *returnType*.

If you omit **As** *returnType* and the function name has no data type suffix character appended, the function returns a value of the data type specified by a Def*type* statement that applies to the function name. A C function can't return a Variant; so a DefVar statement can't apply to the function name.

returnType has the form:

[**LMBCS** | **Unicode**] *dataType*

The *dataType* must match the C function return type exactly; no conversion is performed on the return value.

The optional LMBCS and Unicode keywords may be used with the String data type only, to specify the character set. See the usage information and examples that follow.

Usage

The Public keyword cannot be used in a product object script or %Include file in a product object script, except to declare class members. You must put such Public declarations in (Globals).

You can only declare external functions at the module level. If a function is not typed with a return type or a data type suffix character, LotusScript generates an error.

Passing arguments

By default, LotusScript passes arguments to external functions by reference. Arguments can be passed by value using the ByVal keyword, but only if LotusScript can convert the value passed to the data type of the corresponding C function argument.

Arrays, type variables, and user-defined objects must be passed by reference.

You can't pass lists as arguments to C functions.

You can't use a fixed-length String as an argument.

Product objects can be passed by reference (passing a reference to the instance handle) or by value (passing the instance handle itself).They can be passed by value only by using the keyword ByVal. Parentheses can't be used on the actual argument.

An argument can be typed as Any to avoid data type restrictions. Arguments of type Any are always passed by reference, regardless of the type of data they contain. You can pass a Variant containing an array or list to a C function argument declared as Any.

Using LMBCS or Unicode strings

Use the optional keywords LMBCS and Unicode with a String *argument* or *returnType* to specify the character set.

Unicode designates a Unicode string of two-byte characters (words) using the platform-native byte order.

LMBCS designates a LMBCS optimization group 1 string (multibyte characters).

If neither LMBCS nor Unicode is specified, the string variable uses the platform-native character set.

Calling exported library functions in 32-bit versions of Windows

If you're using a 32-bit version of Windows, the names of exported library functions are case-sensitive; however, LotusScript automatically converts them to uppercase in the Declare statement. To successfully call an exported library function, use the Alias clause to specify the function name with correct capitalization (LotusScript leaves the alias alone).

Examples: Declare statement (external C calls)

Example 1

```
Dim strOut As String
' Declare the external function StrUpr, defined in the library StrLib.
Declare Function StrUpr Lib "StrLib" (ByVal inVal As String) As String
' Call StrUpr
strOut$ = StrUpr("abc")
```

Example 2

```
' Declare an exported library function (SendDLL) with an alias
' to preserve case sensitivity.
Declare Function SendDLL Lib "C:\myxports.dll" _
   Alias "_SendExportedRoutine" (i1 As Long, i2 As Long)
' Call SendDLL
SendDLL(5, 10)
```

Example 3

```
' Pass the string argument amIStr to the function StrFun as
' a Unicode string. The function's return value is also
' a Unicode string.
Declare Function StrFun Lib "lib.dll" _
   (amIStr As Unicode String) As Unicode String
```

Example 4

```
' Pass the string argument amLStr to the function StrFun as
' a LMBCS string. The function's return value is a LotusScript
' platform-native string.
Declare Function StrFun Lib "lib.dll" _
   (amLStr As LMBCS String) As String
```

Declare statement (Forward reference)

Declares a forward reference to a procedure (a function, sub, or property), allowing calls to a procedure that has not yet been defined.

Syntax

Declare [**Static**] [**Public** | **Private**] *procType procName* [**(** [*argList*] **)**] [**As** *returnType*]

Elements

Static

Optional. Specifies that the values of the procedure's local variables are saved between calls to the procedure.

If this keyword is present, it must also be present in the definition of the procedure.

Public | **Private**

Optional. Public indicates that the declared procedure is visible outside this module, for as long as the module is loaded. If this keyword is present, it must also be present in the definition of the procedure.

Private indicates that the declared procedure is visible only within this module. If this keyword is present, it must also be present in the definition of the procedure.

procType

One of the following four keyword phrases, to identify the kind of procedure:

Function
Sub
Property Get
Property Set

procName

The name of a function, sub, or property. If *procType* is Function (a function is being declared), then *procName* can have a data type suffix character appended to declare the type of the function's return value.

argList

A comma-separated list of argument declarations for the procedure. The procedure must be a function or a sub (*procType* must be Function or Sub). The argument declarations must match the argument declarations in the function or sub definition exactly.

The syntax for each argument declaration is:

[**ByVal**] *argument* [**( )** | **List**] [**As** *type*]

ByVal means that *argument* is passed by value: that is, the value assigned to *argument* is a local copy of a value in memory, rather than a pointer to that value.

*argument***()** is an array variable. *argument* **List** identifies *argument* as a list variable. Otherwise, *argument* can be a variable of any of the other data types that LotusScript supports.

As *dataType* specifies the variable's data type. You can omit this clause and use a data type suffix character to declare the variable as one of the scalar data types. If you omit this clause and *argument* doesn't end in a data type suffix character (and isn't covered by an existing Def*type* statement), its data type is Variant.

Enclose the entire list of argument declarations in parentheses.

returnType

The data type of the function's return value. This is optional for a function, and not allowed for a sub or a property, because they don't return values. *returnType* must match the return type specified in the function definition; no conversion is performed on the return value.

If you omit **As** *returnType*, the function name's data type suffix character appended to *procName* (the function name) determines the return value's type. Do not specify both a *returnType* and a data type suffix character.

If you omit **As** *returnType* and *procName* has no data type suffix character appended, the function returns a value either of data type Variant or of the data type specified by a Def*type* statement.

Usage

The IDE implicitly generates forward declarations of procedures; directly entering them in the IDE is unnecessary and causes syntax errors. You can %Include a file containing forward declarations of procedures contained in the file. You can directly enter in the IDE forward declarations of class properties and methods.

The Public keyword cannot be used in a product object script or %Include file in a product object script, except to declare class members. You must put such Public declarations in (Globals).

You can make a forward declaration only at the module level or within a class.

The procedure, if it exists, must be defined in the same scope as the forward declaration. LotusScript does not generate an error if a procedure has a forward declaration but is not defined. (An error will be generated if you try to call a procedure that has been declared but not defined.)

A procedure declared within a class definition cannot be declared as Static.

The use of Static, Public, and Private keywords in a Property Get forward declaration must match their use in the corresponding Property Set forward declaration, if one exists.

Examples: Declare statement (forward reference)

```
' In this example, the forward declaration of the function Times
' allows the use of Times within the definition of the sub PrintFit.
' The function definition of Times appears later in the script.

' Forward declare the function Times.
Declare Function Times (a As Single, b As Single) As Single

' Define the sub PrintFit. It calls Times.
Sub PrintFit (lead As String, x As Single)
   Print lead$, Times (x!, x!)
End Sub

' Define Times.
Function Times (a As Single, b As Single) As Single
   Times = (a! - 1.0) * (b! + 1.0)
End Function

' Call the sub PrintFit.
PrintFit "First approximation is:", 13
' Prints "First approximation is:     168"
```

Deftype statements

Set the default data type for variables, functions, and properties whose names begin with one of a specified group of letters.

Syntax

DefCur *range* [**,** *range*] ...

DefDbl *range* [**,** *range*] ...

DefInt *range* [**,** *range*] ...

DefLng *range* [**,** *range*] ...

DefSng *range* [**,** *range*] ...

DefStr *range* [**,** *range*] ...

DefVar *range* [**,** *range*] ...

Elements

range

A single letter, or two letters separated by a hyphen. Spaces or tabs around the hyphen are ignored. A two-letter range specifies the group of letters including the given letters and any letters between. These must be letters with ASCII code less than 128.

Letters in *range* are case-insensitive. For example, the group of letters J, j, K, k, L, and l can be designated by any one of these *range* specifications: J-L, L-J, j-L, L-j, J-l, l-J, j-l, or l-j.

Usage

The following table lists the Def*type* statements, the data type that each one refers to, and the data type suffix character for that data type.

Statement	*Data type*	*Suffix character*
DefCur	Currency	@
DefDbl	Double	#
DefInt	Integer	%
DefLng	Long	&
DefSng	Single	!
DefStr	String	$
DefVar	Variant	(none)

Def*type* statements can only appear at the module level, but they affect all declarations contained within the module at module level and within its procedures. They do not affect the declarations of data members of types and classes. (They do affect declarations of function members and property members of classes.)

All Def*type* statements in a module must appear before any declaration, explicit or implicit, in the module. Exception: the declaration of a constant (by the Const statement) is not affected by Def*type* statements.

No range in any Def*type* statement can overlap any other range in the same Def*type* statement or in any other Def*type* statement in the same module.

The range A-Z is special. It includes all international characters, not only the letters with ASCII code less than 128. It is the only range specification that includes international characters. For example, to change the default data type of all variables, functions, and properties to Single (the standard data type for several versions of BASIC), specify DefSng A-Z.

Declarations that are explicit as to data type (such as Dim X As Integer, Dim Y$, or Define MyFunction As Double) take precedence over Def*type* declarations.

Examples: Deftype statements

```
DefInt a-z

' x is declared explicitly, with no type.
Dim x
Print TypeName(x)                ' Output: INTEGER

' Ñ is declared explicitly, with no type.
Dim Ñ
Print TypeName(Ñ)                ' Output: INTEGER

' y is declared explicitly, with the String type.
' The specified type overrules the DefInt statement.
Dim y As String
Print TypeName(y)                ' Output: STRING

' b is declared implicitly, with the String type.
' The suffix character $ overrules the DefInt statement.
b$ = "Rebar"
Print TypeName(b$)               ' Output: STRING

' sNum is declared implicitly, which makes it Integer by
' default because DefInt a-z is in effect.
sNum = 17.6
Print TypeName(sNum), sNum       ' Output: INTEGER  18
                                 ' because LotusScript rounds when
                                 ' converting to type Integer.
```

Delete statement

Executes an object's Delete sub, if the sub exists, and then deletes the object.

Syntax

Delete *objRef*

Elements

objRef

An object reference variable or Variant containing an object reference.

Usage

The Delete statement calls the Delete sub in the object's class definition (if one exists), and then sets all references to the object to NOTHING.

If the object's class is a derived class, LotusScript executes the base class's Delete sub (if one exists) after executing the class's Delete sub.

For product objects, the interpretation of a Delete statement is up to the product. In some cases, for example, the Delete statement deletes the object reference, but not the object itself. A product may provide its own script mechanism for deleting the object.

In Lotus Notes Release 4, for example, you can use the Delete statement to delete an object reference to a Notes database, but you use the NotesDatabase class Remove method to delete the database itself (a .nsf file).

Examples: Delete statement

```
' Define the class Customer.
Class Customer
   Public Name As String
   Public Address As String
   Public Balance As Currency

   ' Define a constructor sub for the class.
   Sub New (Na As String, Addr As String, Bal As Currency)
      Me.Name$ = Na$
      Me.Address$ = Addr$
      Me.Balance@ = Bal@
   End Sub

   ' Define a destructor sub for the class.
   Sub Delete
      Print "Deleting customer record for: "; Me.Name$
   End Sub
End Class

' Create an object of the Customer class.
Dim X As New Customer ("Acme Corporation", _
   "55 Smith Avenue, Cambridge, MA", 14.92)
Print X.Balance@
' Output:
' 14.92

' Delete the object, first running the destructor sub.
Delete X
' Output:
' Deleting customer record for: Acme Corporation."

' Then the object is deleted.
```

Dim statement

Declares variables.

Syntax

{ **Dim** | **Static** | **Public** | **Private** } *variableDeclaration* [**,** *variableDeclaration*]...

Elements

Dim | **Static** | **Public** | **Private**

Variable declarations begin with one of the words Dim, Static, Private, or Public.

Dim indicates that a variable is nonstatic and private by default.

- Static indicates that the variable's value is saved between calls to the procedure where the variable is declared.
- Public indicates that the variable is visible outside the scope (module or class) where the variable is defined, for as long as this module remains loaded.
- Private indicates that the variable is visible only within the current scope.

You can use the Static keyword in procedure scope, but not in module or class scope. You can use the Public and Private keywords in module or class scope, but not in procedure scope.

variableDeclaration

The declaration has one of the following forms, depending on the kind of variable being declared:

- Scalar variable: *variableName*[*dtSuffix*] [**As** *type*]
- Object reference variable: *variableName* **As** [**New**] *type* [*argList*]
- List variable: *variableName*[*dtSuffix*] **List** [**As** *type*]
- Array variable: *variableName*[*dtSuffix*] **(** [*bounds*] **)** [**As** *type*]

You can declare any number of variables in a single statement, separated by commas.

variableName

The name of the variable being declared.

dtSuffix

Optional. A character that specifies the data type of *variableName*. The data type suffix characters and the data types that they represent are: @ for Currency, # for Double, % for Integer, & for Long, ! for Single, and $ for String.

type

Optional for scalar variables, lists, and arrays. A valid LotusScript data type, user-defined data type, user-defined class, or product class. This specifies the type of *variableName*.

If *type* is the name of a class, *variableName* is an object reference for that type: its value can only be a reference to an instance of that class or to an instance of a derived class of that class, or the value NOTHING.

New

Optional. Valid only if *type* is the name of a user-defined or product class. New creates a new object of the class named by *type*, and assigns a reference to that object in *variableName*.

Note that in some cases, Lotus products provide other mechanisms for creating product objects in scripts, such as product functions or product object methods. See your Lotus product documentation for details.

argList

Optional. This is valid only if the New keyword is specified.

For user-defined classes, *argList* is the comma-separated list of arguments required by the class constructor sub New, defined in the class named by *type*. For product classes, consult the product documentation.

bounds

Optional. *bounds* is a comma-separated list of bounds for the dimensions of a fixed array. Each bound is specified in the form

[*lowerBound* **To**] *upperBound*

where *lowerBound* is a number designating the minimum subscript allowed for the dimension, and *upperBound* is a number designating the maximum. If no *lowerBound* is specified, the lower bound for the array dimension defaults to zero (0), unless the default lower bound has been changed to 1 using the Option Base statement.

If you don't define any *bounds*, the array is defined to be a dynamic array.

Usage

The Public keyword cannot be used in a product object script or %Include file in a product object script, except to declare class members. You must put such Public declarations in (Globals).

Explicit declarations and implicit declarations

You can declare a variable name either explicitly or implicitly. The Dim statement declares a name explicitly. A name is declared implicitly if it is used (referred to) when it has not been explicitly declared, or when it is not declared as a Public name in another module being used by the module where the name is referred to. You can prohibit implicit declarations by including the statement Option Declare in your script.

Specifying the data type

Either *dtSuffix* or As *type* can be specified in *variableDeclaration*, but not both. If neither is specified, the data type of *variableName* is Variant.

The data type suffix character, if it is specified, is not part of the variable name. When the name is used (referred to) in the script, it can be optionally suffixed by the appropriate data type suffix character.

Declaring arrays

For a fixed array, Dim specifies the type of the array, the number of dimensions of the array, and the subscript bounds for each dimension. Dim allocates storage for the array elements and initializes the array elements to the appropriate value for that data type (see "Initializing variables," later in this section).

For a dynamic array, Dim only specifies the type of the array. The number of dimensions of the array and the subscript bounds for each dimension are not defined; and no storage is allocated for the array elements. The declaration of a dynamic array must be completed by a later ReDim statement.

Arrays can have up to 8 dimensions.

Array subscript bounds must fall in the range -32,768 to 32,767, inclusive.

Declaring lists

A list is initially empty when it is declared: it has no elements, and no storage is allocated for it. An element is added to a list when the list name with a particular list tag first appears on the left-hand side of an assignment statement (a Let statement or a Set statement).

If the character set is single byte, Option Compare determines whether list names are case sensitive. For example, if Option Compare Case is in effect, the names "ListA" and "Lista" are different; if Option Compare NoCase is in effect, these names are the same. If the character set is double-byte, list names are always case and pitch sensitive.

Declaring object reference variables

If *type* is the name of a class and the keyword New is not specified, the initial value of the declared object reference variable is NOTHING. To assign another value to an object reference variable, use the Set statement later in the script.

Dim *variableName* As New *className* generates executable code. When you save a compiled module, module-level executable code is not saved, so be careful about using such a statement at the module level. Your Lotus product may prohibit you from placing executable statements at the module level.

You may prefer to declare the object reference variable at the module level with Dim *variableName* As *className*, which is not executable code, then use a Set statement (which is executable code) in a procedure to bind the object reference variable to an object.

The New keyword is not valid in an array declaration or a list declaration.

Initializing variables

Declaring a variable also initializes it to a default value.

- Scalar variables are initialized according to their data type:
  - Numeric data types (Integer, Long, Single, Double, Currency): Zero (0)
  - Variants: EMPTY
  - Fixed-length strings: A string filled with the NULL character Chr(0).
  - Variable-length strings: The empty string ("").
- Object reference variables are initialized to NOTHING, unless New is specified in the variable declaration.
- Each member of a used-defined data type variable is initialized according to its own data type.
- Each element of an array variable is initialized according to the array's data type.
- A list variable has no elements when it is declared, so there is nothing to initialize.

Visibility of declarations

The default visibility for a declaration at the module level is Private, unless Option Public has been specified.

The default visibility for a variable declaration within a class is Private.

Public and Private can only be used to declare variables in module or class scope. Variables declared within a procedure are automatically Private; members of used-defined data types are automatically Public (these cannot be changed).

Examples: Dim statement

Example 1

```
' Declare a one-dimensional Integer array and a Single variable.
Dim philaMint(5) As Integer
Dim x As Single
x! = 10.0
philaMint%(0) = 3                ' Assigns an Integer value
philaMint%(1) = x                ' Converts Single 10.0 to Integer 10
Print DataType(philaMint%(0)); DataType(philaMint%(1))
' Output:
' 2  2
' Both values are Integers.
```

Example 2

```
Dim xB As New Button("Merge", 60, 125)
```

xB is declared as an object reference variable to hold references to objects of the class named Button. A new Button object is created. For this example, suppose that the constructor sub for the class Button takes three arguments: a name for a button, and x- and y-position coordinates for the location of the button. The new button created is named "Merge," and positioned at (60, 125). A reference to this button is assigned to xB.

Example 3

```
' Declare iVer and kVer as Integer variables. Note that the phrase
' As Integer must be repeated to declare both variables as Integer.
Dim iVer As Integer, kVer As Integer
' Declare nVer as an Integer variable.
' The declared type of mVer is Variant, the default data type, because
' no data type is declared for mVer: there is no As type phrase for
' mVer, and no data type suffix attached to mVer.
Dim mVer, nVer As Integer
Print TypeName(mVer), TypeName(nVer%)         ' Prints EMPTY INTEGER
```

Example 4

```
' Declare marCell and perDue as Integer variables.
' The phrase As Integer declares marCell as an Integer variable.
' The data type suffix % declares perDue as an Integer variable.
Dim marCell As Integer, perDue%
Print TypeName(marCell), TypeName(perDue%)    ' Prints INTEGER INTEGER
```

Example 5

```
Dim marCell% As Integer
' Error, because the Dim statement attempts to declare the
' Integer variable marCell using both the data type suffix character
' for Integer, and the data type name Integer. The declaration should
' include one or the other, but not both.
```

Example 6

```
' A data type suffix character is optional in references to a
' declared variable.

' Declare marCell as an Integer variable.
Dim marCell As Integer
' Use the data type suffix character in a reference to marCell.
marCell% = 1
' Refer to marCell without using the suffix character.
Print marCell                                 ' Prints 1
```

Example 7

```
' Declare marCell as an Integer variable.
Dim marCell As Integer
' Assign integer value to marCell.
marCell% = 1
Print marCell$
' Error, because the Print statement refers to marCell using the
' data type suffix character $ for a String variable, but marCell was
' declared as an Integer.
```

Example 8

```
Dim Leads As New DB ("LotusFormLeads")
```

This Dim *objRef* As New *prodClass*(*argList*) statement declares an object reference to, and creates an instance of, the Lotus Forms DB class. The Dim statement for creating a DB object requires one string argument: a DB object name.

Dir function

Returns file or directory names from a specified directory, or returns a drive volume label.

Syntax

Dir[**$**] [**(** *fileSpec* [**,** *attributeMask*] **)**]

Elements

fileSpec

A string expression that specifies a path and the file names you want returned. The argument is required only for the first call to Dir$ for any path.

Standard wildcard characters can be used in *fileSpec* to designate all files satisfying the wildcard criterion. Asterisk (*) for either the file name or the extension designates all files with any characters in that position. Question mark (?) in any character position in either part of the name designates any single character in that position.

attributeMask

An integer expression whose value specifies what names should be returned. If this argument is omitted, the names of normal files that match *fileSpec* are returned. If you supply an *attributeMask* argument, you must supply a *fileSpec* argument.

To include other files in the returned list of file names, specify the sum of those values in the following table that correspond to the desired kinds of files:

Mask	*File attribute*	*Constant*
0	Normal file	ATTR_NORMAL
2	Hidden file	ATTR_HIDDEN
4	System file	ATTR_SYSTEM
8	Volume label	ATTR_VOLUME. If this is specified, then the presence (or absence) of 2, 4, and 16 is irrelevant. The hidden, system, or directory attributes are not meaningful for a volume label.
16	Directory	ATTR_DIRECTORY

Return value

Dir returns a Variant of DataType 8 (String), and Dir$ returns a String.

Usage

The constants in the table are defined in the file lsconst.lss. Including this file in your script allows you to use constant names instead of their numeric values.

To determine whether a particular file exists, use an exact file name for the *file_spec* argument to Dir or Dir$. The return value is either the file name, or, if the file does not exist, the empty string (“”).

The first call to Dir or Dir$ returns the name of the first file in the specified directory that fits the file name specifications in the *fileSpec* argument. Then:

- Subsequent calls to Dir or Dir$ without an argument retrieve additional file names that match *fileSpec*.
- If there are no more file names in the specified directory that match the specification, Dir returns a Variant of DataType 8 (String); Dir$ returns the empty string (“”).

If Dir or Dir$ is called without an argument after the empty string has been returned, LotusScript generates an error.

You can call the Dir function with no arguments as either Dir or Dir( ). You can call the Dir$ function with no arguments as either Dir$ or Dir$()

Examples: Dir function

```
' List the contents of the c:\ directory, one entry per line.
Dim pathName As String, fileName As String
pathName$ = "c:\*.*"
fileName$ = Dir$(pathName$, 0)

Do While fileName$ <> ""
   Print fileName$
   fileName$ = Dir$()
Loop
```

Do statement

Executes a block of statements repeatedly while a given condition is true, or until it becomes true.

Syntax 1

Do [**While** | **Until** *condition*]

    [*statements*]

Loop

Syntax 2

Do

    [*statements*]

Loop [**While** | **Until** *condition*]

Elements

condition

Any numeric expression. 0 is interpreted as FALSE, and any other value is interpreted as TRUE.

Usage

In Syntax 1, *condition* is tested before entry into the loop, and before each subsequent repetition. The loop repeats as long as *condition* evaluates to TRUE (if you specify **While**), or until *condition* evaluates to TRUE (if you specify **Until**).

In Syntax 2, *condition* is tested after the body of the loop executes once, and after each subsequent repetition. The loop repeats as long as *condition* evaluates to TRUE (if you specify **While**), or until *condition* evaluates to TRUE (if you specify **Until**).

Terminating the loop

You can exit the loop with an Exit Do statement or a GoTo statement. Exit Do transfers control to the statement that follows the Do...Loop block; GoTo transfers control to the statement at the specified label.

If you do not specify a While or Until *condition*, the loop will run forever or until an Exit Do or a GoTo statement is executed within the loop. For example, this loop executes forever:

```
Do
    ' ...
Loop
```

Examples: Do statement

```
' Each loop below executes four times,
' exiting when the loop variable reaches 5.
Dim i As Integer, j As Integer
i% = 1
j% = 1
Do While i% < 5          ' Test i's value before executing loop.
   i% = i% + 1
   Print i% ;
Loop
' Output:
' 2  3  4  5
Do
   j% = j% + 1
   Print j% ;
Loop Until j% >= 5       ' Test j's value after executing loop.
' Output:
' 2  3  4  5
```

Dot notation

Use dot notation to refer to members of user-defined types, user-defined classes, and product classes.

Syntax 1

*typeVarName***.***memberName*

Syntax 2

*objRefName***.***memberName* [**(***argList***)**]

Elements

typeVarName

A variable of a user-defined data type.

memberName

A member of a user-defined type, a user-defined class, or a product class. Class members may include methods, properties, and variables.

objRefName

An object reference variable.

argList

Optional. A list of one or more arguments; some class methods and properties require an argument list.

Usage

Use dot notation to refer to the members of user-defined data types, user-defined classes, and product classes.

When referring to the currently selected product object, you may omit *objRefName*. In some cases, you can use bracket notation, substituting **[***prodObjName***]** for *objRefName*. For more information, see your Lotus product documentation.

Note that dot notation is interpreted differently when it appears within a With statement. See that topic for details.

Examples: Dot notation

Lotus Forms example

In Lotus Forms, you use the Forms Designer to place pictures on a form. A picture is an instance of the Forms Picture class. In a script, you can change the bitmap or metafile that the Picture object displays.

This example sets the value of the FileName property and uses the Update method to refresh the display. The FileName property and Update method both apply to the Picture class. For information about Lotus Forms classes, see the Lotus Forms documentation.

```
' statePicture is an object reference to a Picture object.
If state$ = "Ohio" Then
   ' Set the FileName property and refresh the display.
   statePicture.FileName = "c:\maps\ohio.bmp"
   statePicture.Update
End If
```

Double data type

Specifies a variable that contains a double-precision floating point value maintained as an 8-byte floating point value.

Usage

The Double suffix character for implicit type declaration is #.

Double variables are initialized to 0.

The range of Double values is -1.7976931348623158E+308 to 1.7976931348623158E+308, inclusive.

On UNIX platforms, the range is -1.7976931348623156E+308 to 1.797693134862315E+308, inclusive.

The smallest non-zero Double value (disregarding sign) is 2.2250738585072014E-308.

LotusScript aligns Double data on an 8-byte boundary. In user-defined types, declaring variables in order from highest to lowest alignment boundaries makes the most efficient use of data storage space.

Examples: Double data type

```
' Explicitly declare a Double variable.
Dim rate As Double
rate# = .85

' Implicitly declare a Double variable.
interest# = rate#

Print interest#                         ' Prints .85
```

End statement

Terminates execution of the currently executing script.

Syntax

End [*returnCode*]

Elements

returnCode

Optional. An integer expression. The script returns the value of this expression to the Lotus product that executed the script.

Usage

Some Lotus products do not expect a return value when an End statement executes. See the product's documentation. If the product does not expect a return value, you do not need to use *returnCode*. (The product will ignore it if you do.)

Examples: End statement

```
' The End statement terminates execution of the script
' that is running when the function is called.
Function Func1 ()
   Print 1
   End                          ' Terminates program execution
   Print 2                      ' Never executed
End Function                    ' Ends the function definition
Func1
' Output:
' 1
```

Environ function

Returns information about an environment variable from the operating system.

Syntax 1

Environ[**$**] **(** { *environName* | *n* } **)**

Elements

environName

A string of uppercase characters indicating the name of an environment variable.

n

A numeric value from 1 to 255, inclusive, indicating the position of an environment variable in the environment string table.

Return value

Environ returns a Variant, and Environ$ returns a String.

If you specify the environment variable by name with *environName*, LotusScript returns the value of the specified environment variable. If that environment variable is not found, LotusScript returns the empty string (“”). If *environName* is the empty string or evaluates to NULL or EMPTY, LotusScript generates an error.

If you specify the environment variable by position with *n*, LotusScript returns the entire environment string, including the name of the environment variable. If *n* is larger than the number of strings in the environment string table, LotusScript returns the empty string (“”).

If *n* is less than 1, greater than 255, an EMPTY Variant, or NULL, LotusScript generates an error.

Examples: Environ function

Microsoft Windows 3.1 stores temporary files in the directory defined by the Temp environment variable. This example makes the temp directory the current directory, and writes the string you enter to a file (MYAPP.TMP) in that directory. To determine the location of your temp directory, see the Set Temp command in your AUTOEXEC.BAT.

```
Dim TempDir As String, tempFile As Integer
Dim tempFileName As String, tempStuff As String
tempStuff$ = InputBox("Enter some temporary information")
TempDir$ = Environ("Temp")
ChDir TempDir$
tempFile% = FreeFile()
tempFileName$ = "myapp.tmp"
Open tempFileName$ For Output As tempFile%
Print #tempFile%, tempStuff$
Close tempFile%
```

EOF function

Returns an integer value that indicates whether the end of a file has been reached.

Syntax

EOF (*fileNumber* **)**

fileNumber

The ID number assigned to the file when it was opened.

Return value

The return value depends on the type of file that you are using. The following table shows the EOF return values for binary, random, and sequential file types.

File type	*EOF returns TRUE (-1) if:*	*EOF returns FALSE (0) if:*
Binary	The last executed Get statement cannot read the amount of data (the number of bytes) requested.	It successfully reads the amount of data requested.
Random	The last executed Get statement cannot read an entire record.	It successfully reads an entire record.
Sequential	The end of the file has been reached.	The end of the file has not been reached.

Usage

The end of file is determined by the operating system (from the file length stored in the file system). A Ctrl+Z character (ASCII 26) is not considered an end-of-file marker for any type of file: sequential, random, or binary.

Examples: EOF function

```
' Open a file, print it, and close the file.
Dim text As String, fileNum As Integer
fileNum% = FreeFile()

Open "c:\config.sys" For Input As fileNum%
Do Until EOF(1)
   Line Input #1, text$
   Print text$
Loop
Close fileNum%
```

Erase statement

Deletes an array, list, or list element.

Syntax

Erase { *arrayName* | *listName* | *listName* **(** *tag* **)** }
[**,** { *arrayName* | *listName* | *listName* **(** *tag* **)** }]...

Elements

arrayName

An array or a Variant variable containing an array. *arrayName* can end with empty parentheses.

listName

A list or a Variant variable containing a list. *listName* can end with empty parentheses.

tag

The list tag of a list element to be erased from the specified list.

Usage

The following table shows how the Erase statement affects arrays and lists.

Item	*Effect of Erase statement*
Fixed array	Its elements are reinitialized.
Dynamic array	LotusScript removes all elements from storage and recovers the storage. The array retains its type, but has no elements. You must use ReDim to redeclare the array before referring to its elements again. If you used ReDim before it was erased, the array maintains the same number of dimensions.
List	LotusScript removes all elements from storage and recovers the storage. The list retains its type, but has no elements.
List element	The element no longer exists in the list.

Examples: Erase statement

```
' Use Erase to reinitialize the Integer elements of the
' array baseLine to zero.
Option Base 1
Dim baseLine(3) As Integer       ' Declare the fixed array baseLine.
baseLine%(1) = 1                 ' Assign values to baseLine.
baseLine%(2) = 2
baseLine%(3) = 6
Erase baseLine%                  ' Erase baseLine.
Print baseLine%(1)              ' Prints 0.
```

Erl function

Returns the line number in the script source file where the current error occurred.

Syntax
Erl

Return value
Erl returns an Integer. It returns FALSE (0) if there is no current error, which signifies that the most recent error has been handled.

Usage
You can call the function as either Erl or Erl().

The line number returned by Erl is for the procedure handling the error. If a calling procedure contains an On Error statement and the called procedure does not, an error in the called procedure is reported at the line number of the Call statement or function reference in the calling procedure.

Examples: Erl function
```
' Assign the line number where an error occurs
' (if any) to the variable x.
Sub RepErr
   Dim x As Integer
   x% = Erl()
   Print x%
End Sub
Call RepErr
' Output:
' 0                          ' There is no current error.
```

Err function

Returns the current error number.

Syntax
Err

Return value
Err returns an Integer. If there is no current error, Err returns FALSE (0).

Usage

The error number is set when an error occurs, or by the Err statement. Generally, the function Err is used within an error-handling routine.

You can call the function as either Err or Err().

Examples: Err function

```
' This example uses the Err function, Err statement, Error function,
' and Error statement.
' Ask the user to enter a number between 1 and 100. If the user's
entry
' cannot be converted to a 4-byte single, an error occurs.
' The example defines two additional errors for numeric entries
' not in the range 1 - 100.

Public x As Single
Const TOO_SMALL = 1001, TOO_BIG = 1002
Sub GetNum
   Dim Num As String
   On Error GoTo Errhandle
   Num$ = InputBox$("Enter a value between 1 and 100:")
   x! = CSng(Num$)           ' Convert the string to a 4-byte single.
   ' Check the validity of the entry.
   If x! < 1 Then
      Error TOO_SMALL, "The number is too small or negative."
   ElseIf x! > 100 Then
      Error TOO_BIG, "The number is too big."
   End If
   ' If the script gets here, the user made a valid entry.
   MessageBox "Good job! " & Num$ & " is a valid entry."
   Exit Sub
   ' The user did not make a valid entry.
   ' Display the error number and error message.
Errhandle:
   ' Use the Err function to return the error number and
   ' the Error$ function to return the error message.
   MessageBox "Error" & Str(Err) & ": " & Error$
   Exit Sub
End Sub
GetNum                        ' Call the GetNum sub.
```

Err statement

Sets the current error number.

Syntax

Err = *errNumber*

Elements

errNumber

A numeric expression whose value is an error number.

Usage

The Err statement sets the current error number to an error number you specify. This may be any number in the range 0 to 32767 inclusive.

Examples: Err statement

```
' This example uses the Err function, Err statement, Error function,
' and Error statement.
' Ask the user to enter a number between 1 and 100. If the user's
entry
' cannot be converted to a 4-byte single, an error occurs.
' the example defines two additional errors for numeric entries
' not in the range 1 - 100.

Public x As Single
Const TOO_SMALL = 1001, TOO_BIG = 1002
Sub GetNum
   Dim Num As String
   On Error GoTo Errhandle
   Num$ = InputBox$("Enter a value between 1 and 100:")
   x! = CSng(Num$)           ' Convert the string to a 4-byte single.
   ' Check the validity of the entry.
   If x! < 1 Then
      Error TOO_SMALL, "The number is too small or negative."
   ElseIf x! > 100 Then
      Error TOO_BIG, "The number is too big."
   End If
   ' If the script gets here, the user made a valid entry.
   MessageBox "Good job! " & Num$ & " is a valid entry."
   Exit Sub
   ' The user did not make a valid entry.
   ' Display the error number and error message.
Errhandle:
   ' Use the Err function to return the error number and
   ' the Error$ function to return the error message.
   MessageBox "Error" & Str(Err) & ": " & Error$
   Exit Sub
End Sub
GetNum                       ' Call the GetNum sub.
```

Error function

Returns an error message for either a specified error number or the current error.

Syntax

Error[**$**] [**(** *errNumber* **)**]

Elements

errNumber

A numeric expression whose value is an error number. If no *errNumber* is specified, LotusScript returns the message for the current (most recent) error.

Return value

Error returns a Variant, and Error$ returns a String. If no *errNumber* is specified, and there is no current error, the function returns the empty string (“”).

You can call the Error function with no arguments as either Error or Error( ). You can call the Error$ function with no arguments as either Error$ or Error$( ).

Examples: Error function

```
' This example uses the Err function, Err statement, Error function,
' and Error statement.
' Ask the user to enter a number between 1 and 100. If the user's
entry
' cannot be converted to a 4-byte single, an error occurs.
' The example defines two additional errors for numeric entries
' not in the range 1 - 100.

Public x As Single
Const TOO_SMALL = 1001, TOO_BIG = 1002
Sub GetNum
   Dim Num As String
   On Error GoTo Errhandle
   Num$= InputBox$("Enter a value between 1 and 100:")
   x! = CSng(Num$)           ' Convert the string to a 4-byte single.
   ' Check the validity of the entry.
   If x! < 1 Then
      Error TOO_SMALL, "The number is too small or negative."
   ElseIf x! > 100 Then
      Error TOO_BIG, "The number is too big."
   End If
   ' If the script gets here, the user made a valid entry.
   MessageBox "Good job! " & Num$ & " is a valid entry."
   Exit Sub
   ' The user did not make a valid entry.
   ' Display the error number and error message.
```

```
Errhandle:
   ' Use the Err function to return the error number and
   ' the Error$ function to return the error message.
   MessageBox "Error" & Str(Err) & ": " & Error$
   Exit Sub
End Sub
GetNum                        ' Call the GetNum sub.
```

Error statement

Signals an error number and its corresponding message.

Syntax

Error *errNumber* [**,** *msgExpr*]

Elements

errNumber

A numeric expression whose value is a LotusScript-defined error number or a user-defined error number. The *errNumber* argument can be any number between 1 and 32767 inclusive.

msgExpr

Optional.

A string expression containing an error message. This string replaces any existing message associated with the error number.

Usage

If *errNumber* is a LotusScript-defined error number, this Error statement simulates a LotusScript error. If it is not, this statement creates a user-defined error. When the Error statement is executed, LotusScript behaves as if a run-time error has occurred. If no error handling is in effect (set up by an On Error statement) for the specified error, execution ends and an error message is generated.

The error message generated is *msgExpr* if it is specified. If *msgExpr* is omitted, the error message is the LotusScript error message for the specified error number, if that number designates a LotusScript error. Otherwise the message "User-defined error" is generated.

Examples: Error statement

```
' This example uses the Err function, Err statement, Error function,
' and Error statement. The On Error statement specifies which error
the
' error-handling routine ErrTooHigh handles. The Error statement tests
' the routine.
' Ask the user to enter a number between 1 and 100. If the user's
entry
' cannot be converted to to a 4-byte single, an erorr occurs.
' The example defines two additional errors for numeric entries not in
' the range 1 - 100.

Public x As Single
Const TOO_SMALL = 1001, TOO_BIG = 1002
Sub GetNum
   Dim Num As String
   On Error GoTo Errhandle
   Num$= InputBox$("Enter a value between 1 and 100:")
   x! = CSng(Num$)           ' Convert the string to a 4-byte single.
   ' Check the validity of the entry.
   If x! < 1 Then
      Error TOO_SMALL, "The number is too small or negative."
   ElseIf x! > 100 Then
      Error TOO_BIG, "The number is too big."
   End If
   ' If the script gets here, the user made a valid entry.
   MessageBox "Good job! " & Num$ & " is a valid entry."
   Exit Sub
   ' The user did not make a valid entry.
   ' Display the error number and error message.
Errhandle:
   ' Use the Err function to return the error number and
   ' the Error$ function to return the error message.
   MessageBox "Error" & Str(Err) & ": " & Error$
   Exit Sub
End Sub
GetNum                       ' Call the GetNum sub.
```

Evaluate function and statement

Execute a Lotus product macro.

Syntax

Evaluate (*macro* [**,** *object*] **)**

Elements

macro

The text of a Lotus product macro, in the syntax that the product recognizes. Refer to the Lotus product documentation for the correct syntax of the macro.

The macro text must be known at compile time, so use a constant or string literal. Do not use a string variable.

object

Optional. The name of a product object. Refer to the product documentation to determine if the macro requires an object, and what the object is.

Return value

If the Lotus product macro being executed returns a value, the Evaluate function returns a Variant containing that value. Otherwise, the function does not return a value.

Examples: Evaluate function and statement

```
' For each document in a Notes database, use a Notes macro to compute
' the average for a list of numeric entries in the NumberList field.
' Evaluate returns a Variant, and Notes macros return an array. In
' this case, the array contains only one element (element 0).
' For more information, see the Notes documentation.

Sub Click(Source As Button)
   ' The macro text must be known at compile time.
   Const NotesMacro$ = "@Sum(NumberList) / @Elements(NumberList)"
   Dim result As Variant, j As Integer
   Dim db As New NotesDatabase("", "MYSALES.NSF")
   Dim dc As NotesDocumentCollection
   Dim doc As NotesDocument
   Set dc = db.AllDocuments
   For j% =  1 To dc.Count
      Set doc = dc.GetNthDocument(j%)
      result = Evaluate(NotesMacro$, doc)
      MessageBox("Average is " & result(0))
   Next
End Sub
```

Execute function and statement

Compiles and executes a text expression as a temporary module.

Statement Syntax
Execute *text*

Function Syntax
Execute (*text* **)**

Elements
text

A string expression specifying the text to be compiled and executed.

Return value
The Execute function returns one of the following values:

- The return code of an End statement, if one was executed.
- Zero (0), if no End statement was executed, or if the executed End statement had no return value.

Usage
LotusScript considers *text* a separate script, compiling and executing it as a temporary module that's unloaded as soon as execution finishes.

Variables declared in the calling script (where the Execute statement appears) are only accessible in the temporary module if they are declared Public. Both these Public variables, and variables declared Public in external modules used by the calling script, will be accessible automatically. To reference a local variable in the temporary module, use the CStr function to convert its value to a string, and then include the result in *text*.

Variables declared in the temporary module are not accessible outside of that script.

To delimit text that spans several lines or includes double-quote characters, use vertical bars (| |) or braces ({ }).

Any compilation error in the temporary module will be reported as a run-time error in the scope containing the Execute statement. Any run-time error in the temporary module will be reported as a run-time error within the scope of that module, not the scope containing the Execute statement. To handle run-time errors within the temporary module, use the On Error statement.

The Execute statement is not legal at the module level; you can use it only in procedures.

Examples: Execute function and statement

Example 1 (Execute statement)

```
' The Execute statement performs a calculation entered by the user and
' displays the result. If the user enters an invalid calculation, a
' compilation error occurs, and the DoCalc sub displays an
' appropriate message. The Option Declare statement disallows
' the implicit declaration of variables in the calculation. The user
' can enter 700 * 32, for example, or "My name is " & "Fred", or
' Today - 365, but an entry such as x / y generates an error.

Sub DoCalc
   ' To handle any compilation error in the Execute statement
   On Error GoTo BadCalc
   Execute |Option Declare
           Dim x ' x is a Variant to accept any calculation.
           x =  | & InputBox ("Enter your calculation") & |
           MessageBox "The result is " & x|
   Exit Sub
' Report an error and exit.
BadCalc:
   MessageBox "Not a valid calculation"
   Exit Sub
End Sub
DoCalc                                  ' Call the sub.
```

Example 2 (Execute function)

```
' You can use the Execute function to return an integer such as a
' status code. In this example, the Execute function performs the
' calculation entered by the user. If the result is less than 0
' or greater than 1 (100%), Execute returns a status code, and the
' ComputeInterest sub displays an appropriate message.

Sub ComputeInterest
   Dim script As String, calc As String, retcode As Integer
   calc$ = InputBox("Compute loan interest (charge/loan)")
   script$ = _
      |Option Declare
      Sub Initialize
         Dim pct As Single
         pct! = | &  calc$ & |
         If pct! < 0 Then
            End -2                      ' -2 is a status code.
         ElseIf pct! > 1 Then
            End -3                      ' -3 is a status code.
         End If
         MessageBox("Interest is " & Format(pct!,"percent"))
      End Sub|
   retcode% = Execute (script$)
   If retcode% = -2 Then
```

```
      MessageBox("You computed a negative interest rate!")
   ElseIf retcode% = -3 Then
      MessageBox("You computed an excessive interest rate!")
   End If
End Sub
ComputeInterest                          ' Call the sub.
```

Exit statement

Terminates execution of the current block statement.

Syntax

Exit *blockType*

Elements

blockType

A keyword designating the type of the block statement for which execution is to be terminated. It must be one of the following keywords:

Do
For
ForAll
Function
Sub
Property

Usage

When LotusScript encounters this statement, it returns control to the scope containing the block statement for which execution is to be terminated.

An Exit statement of a particular type is legal only within an enclosing block statement. LotusScript returns control from the innermost block statement or procedure of that type.

However, the innermost block statement containing the Exit statement need not be of that type. For example, a function definition can include a For...Next block statement, and an Exit Function statement can appear within this statement. If LotusScript encounters the Exit Function statement during execution, control is returned immediately from the function, in which case the For...Next block statement is not executed to completion.

The following table shows the rules for transfer of control after the Exit statement.

Exit block type	*Execution continues*
Exit Do	At the first statement following the end of the Do block statement.
Exit For	At the first statement following the end of the For block statement.
Exit ForAll	At the first statement following the end of the ForAll block statement.
Exit Function	In the calling script, as it would from a normal return from the procedure.
Exit Sub	In the calling script, as it would from a normal return from the procedure.
Exit Property	In the calling script, as it would from a normal return from the procedure.

If you exit a function or a Property Get without assigning a value to the function or property variable, that function or property returns the initialized value of the variable. Depending on the data type of the function or property's return value, this value can be either 0, EMPTY, or the empty string ("").

Examples: Exit statement

```
' The user is asked to enter a 5-character string. If the length of
' the entry is not 5, the result of Exit Function is to return the
' empty string and issue a message telling you the entry is invalid.

Function AssignCode As String
   Dim code As String
   code$ = InputBox("Enter a 5-character code")
   If Len(code$) <> 5 Then Exit Function
   AssignCode = code$               ' It is a valid code.
End Function
If AssignCode() <> "" Then
   MessageBox "You entered a valid code."
Else
   MessageBox "The code you entered is not valid."
End If
```

Exp function

Returns the exponential (base *e*) of a number.

Syntax
Exp (*numExpr* **)**

Elements
numExpr

Any numeric expression, designating the power to which you wish to raise the value *e*.

If the value of *numExpr* exceeds 709.78, LotusScript returns an overflow error.

Return value
Exp returns the exponential (base *e*) of *numExpr*.

The data type of the return value is Double.

Usage
The value of *e* is approximately 2.71828182845905.

Exp is the inverse function of Log.

Examples: Exp function
```
Print Exp(2)                    ' Prints  7.38905609893065
```

FileAttr function

Returns the access type, or the operating system file handle, for an open file.

Syntax
FileAttr (*fileNumber***,** *attribute* **)**

Elements
fileNumber

The number associated with the file when you opened it.

attribute

A number (either 1 or 2) specifying the type of information you want. Instead of 1 or 2, you can specify the constant ATTR_MODE or ATTR_HANDLE, respectively. These constants are defined in the file lsconst.lss. Including this file in your script allows you to use constants instead of their numeric values.

Return value

If *attribute* is ATTR_HANDLE, then FileAttr returns the operating system file handle for the file.

If *attribute* is ATTR_MODE, then FileAttr returns an integer representing the access for the file, as shown in the following table.

Return value	*Access*	*Constant*
1	Input	ATTR_INPUT
2	Output	ATTR_OUTPUT
4	Random	ATTR_RANDOM
8	Append	ATTR_APPEND
32	Binary	ATTR_BINARY

Examples: FileAttr function

```
' The following example creates a file and displays its attributes.

%Include "lsconst.lss"

Dim mode As String, msg As String
Dim hdl As Integer, fileNum As Integer
fileNum% = FreeFile()

Open "data.txt" For Append As fileNum%
hdl% = FileAttr(fileNum%, ATTR_HANDLE)

Select Case FileAttr(fileNum%, ATTR_MODE)
   Case 1 : mode$ = "Input"
   Case 2 : mode$ = "Output"
   Case 4 : mode$ = "Random"
   Case 8 : mode$ = "Append"
   Case 32 : mode$ = "Binary"
End Select

Close fileNum%
Print "DOS File Handle = "; hdl%; "Mode = "; mode$
```

FileCopy statement

Makes a copy of a file.

Syntax

FileCopy *source* **,** *destination*

Elements

source

A string expression containing the name of the file you want to copy. The expression can optionally include a path.

destination

A string expression containing the name to be given to the copy. The expression can optionally include a path.

Usage

The file being copied must not be open.

The *source* and *destination* strings cannot include wildcard characters.

If *destination* names a file that already exists, the copy replaces the existing file with that name. To prevent this, you can use the Dir function to determine whether a file with the name *destination* already exists. Or use the SetFileAttr statement to set the read-only attribute for the file.

Examples: FileCopy statement

```
' Copy C:\WINDOWS\APP.BAT to the root directory of drive C: and
' name the copy APPLOAD.BAT.
FileCopy "C:\WINDOWS\APP.BAT", "C:\APPLOAD.BAT"
```

FileDate time function

Returns a string showing the date and time that a file was created or last modified.

Syntax

FileDateTime (*fileName* **)**

Elements

fileName

A string expression; you can include a path. *fileName* cannot include wildcard characters.

Return value

The returned date and time appear in the default format based on the operating system's international settings. If the file doesn't exist, FileDateTime returns an error.

Examples: FileDateTime function

```
' This script creates a file called data.txt
' and prints its creation date and time.

%Include "lsconst.lss"

Dim fileName As String, fileNum As Integer
fileNum% = FreeFile()
fileName$ = "data.txt"

Open fileName$ For Output As fileNum%   ' Create data.txt file.
Close fileNum%
Print fileName$; " Created on "; FileDateTime(fileName$)
```

FileLen function

Returns the length of a file in bytes.

Syntax

FileLen (*fileName* **)**

Elements

fileName

A string expression; you can optionally include a path. The *fileName* cannot contain wildcard characters.

Return value

FileLen returns a Long value.

Examples: FileLen function

```
' Assign the length (in bytes) of the file c:\config.sys
' to the variable verLen, and print the result.
Dim verLen As Long
verLen& = FileLen("c:\config.sys")
Print verLen&
```

Fix function

Returns the integer part of a number.

Syntax

Fix (*numExpr* **)**

Elements

numExpr

Any numeric expression.

Return value

Fix returns the value of its argument with the fractional part removed. The data type of the return value is determined by the data type of *numExpr*. The following table shows special cases.

numExpr	*Return value*
NULL	NULL
Variant containing a string interpretable as a number	Double
Variant containing a date/time value	The date part of the value

Usage

The Fix function rounds toward 0:

- For a positive argument, Fix returns the nearest integer less than or equal to the argument (if the argument is between 0 and 1, Fix returns 0).
- For a negative argument, Fix returns the nearest integer larger than or equal to the argument (if the argument is between 0 and -1, Fix returns 0).

The Fix function and the Int function behave differently. The return value from Int is always less than or equal to its argument.

Examples: Fix function

```
Dim xF As Integer, yF As Integer
Dim xT As Integer, yT As Integer
xF% = Fix(-98.8)
yF% = Fix(98.2)
xT% = Int(-98.8)
yT% = Int(98.2)
Print xF%; yF%
' Output:
' -98  98
Print xT%; yT%
' Output:
' -99  98
```

For statement

Executes a block of statements a specified number of times.

Syntax

For *countVar* **=** *first* **To** *last* [**Step** *increment*]

[*statements*]

Next [*countVar*]

Elements

countVar

A variable used to count repetitions of the block of statements. The data type of *countVar* should be numeric.

first

A numeric expression. Its value is the initial value of *countVar*.

last

A numeric expression. Its value is the final value of *countVar*.

increment

The value (a numeric expression) by which the *countVar* is incremented after each execution of the statement block. The default value of *increment* is 1. Note that *increment* can be negative.

Usage

After exit from a loop, the *countVar* for the loop has its most recent value.

Executing the loop the first time

Before the block of statements is executed for the first time, *first* is compared to *last*. If *increment* is positive and *first* is greater than *last*, or if *increment* is negative and *first* is less than *last*, the body of the loop isn't executed. Execution continues with the first statement following the For loop's terminator (Next).

Otherwise *countVar* is set to *first* and the body of the loop is executed.

Executing the loop more than once

After each execution of the loop, *increment* is added to *countVar*. Then *countVar* is compared to *last*. When the value of *countVar* is greater than *last* for a positive *increment*, or less than *last* for a negative *increment*, the loop is complete and execution continues with the first statement following the For loop's terminator (Next). Otherwise the loop is executed again.

Exiting the loop early

You can exit a For loop early with an Exit For statement or a GoTo statement. When LotusScript encounters an Exit For, execution continues with the first statement following the For loop's terminator (Next). When LotusScript encounters a GoTo statement, execution continues with the statement at the specified label.

Nested For loops

You can include a For loop within a For loop, as in the following example:

```
Dim x As Integer
Dim y As Integer
For x% = 1 To 3
   For y% = 1 To 2
      Print x% ;
   Next                          ' Next y
Next                             ' Next x
' Output: 1 1 2 2 3 3
```

If you don't include *countVar* as part of a For loop terminator (Next), LotusScript matches For loop delimiters from the most deeply nested to the outermost.

LotusScript lets you combine For loop terminators when they are contiguous, as in the following example:

```
Dim x As Integer
Dim y As Integer
For x% = 1 To 3
   For y% = 1 To 2
      Print x% ;
Next y%, x%           ' Terminate the inner loop and then the outer
loop.
' Output: 1 1 2 2 3 3
```

Examples: For statement

```
' Compute factorials for numbers from 1 to 10
Dim m As Long
Dim j As Integer
m& = 1
For j% = 1 To 10
   m& = m& * j%
   Print m&
Next
' Output:
' 1  2  6  24  120  720  5040  40320  362880  3628800
```

ForAll statement

Executes a block of statements repeatedly for each element of an array, a list, or a collection. A collection is an instance of a product collection class or an OLE collection class.

Syntax

ForAll *refVar* **In** *container*

[*statements*]

End ForAll

Elements

refVar

A reference variable for the array, list, or collection element. In the body of the ForAll loop, you use *refVar* to refer to each element of the array, list, or collection named by *container*. *refVar* can't have a data type suffix character appended.

container

The array, list, or collection whose elements you wish to process.

Usage

On entry to the loop, *refVar* refers to the first element of the array, list, or collection. On each successive iteration, *refVar* refers to the next element of the array, list, or collection. Upon completion of the loop, execution continues with the first statement following the loop's End ForAll statement.

Note If you're using Forall on an array of arrays, do not ReDim the iterator (this generates the "Illegal ReDim" error).

Exiting the loop early

You can force the loop to be exited early with the Exit ForAll statement or the GoTo statement. When LotusScript encounters an Exit ForAll statement, execution immediately continues with the first statement following the loop's terminator (**End ForAll**). When LotusScript encounters a GoTo statement, execution immediately continues with the statement at the specified label.

Using *refVar*

Since *refVar* is an alias for the actual array, list, or collection element, you can change the value of the element to which it refers by assigning a new value to *refVar*. For example:

```
ForAll x In y
   x = x + 1
End ForAll
```

This adds 1 to the value of each element in the array, list, or collection named y.

If *container* is a list, you can pass *refVar* to the ListTag function to get the name (the list tag) of the list element that *refVar* currently refers to. For example:

```
Print ListTag(refVar)
```

Because *refVar* is implicitly defined by the ForAll statement, you should not include it in your variable declarations. The scope of *refVar* is the loop, so you can't refer to it from outside of the loop.

If *container* is an array or list, *refVar* has the data type of the array or list being processed. If this data type cannot be determined by LotusScript at compile time or if *container* is a collection, *refVar* is a Variant. In that case, the data type of the array or list cannot be a user-defined data type, because Variants cannot be assigned values of a user-defined data type.

You can reuse a *refVar* in a subsequent ForAll loop, provided that the data type of the container matches that of the container in the ForAll loop where *refVar* was first defined.

You can't use the ReDim statement on the reference variable. For example, suppose that zArr is an array of arrays, and a ForAll statement begins:

```
ForAll inzArr In zArr
```

Then the statement ReDim inzArr(2) generates an error.

Examples: ForAll statement

Example 1

```
Dim myStats List As Variant
myStats("Name") = "Ian"
myStats("Age") = 29
ForAll x In myStats
   Print ListTag(x); " = "; x
End ForAll
' Output:
' Name = Ian
' Age =  29
```

Example 2

```
Dim minima(5) As Integer
minima%(0) = 5
minima%(1) = 10
minima%(2) = 15
' Set all elements of array minima to 0.
ForAll x In minima%
   x = 0
End ForAll
```

Example 3

In Freelance Graphics, an Application object contains a DocumentCollection object. The DocumentCollection object contains a collection of Document objects. Each Document object contains a PageCollection object. Each PageCollection object contains a number of Page objects. Each Page object contains an ObjectCollection object. ObjectCollection is a heterogenous collection that may include TextBox objects.

In addition to For loops, you can use ForAll loops or indexing to access individual members of a collection class. This example uses three nested ForAll loops to iterate through the collections. Within individual TextBlock objects, the script uses indexing to set list entries at levels 2 through 5 in each TextBox object to Italic.

```
Dim level As Integer
ForAll doc In [Freelance].Documents
   ForAll pg In Doc.Pages
      ForAll obj In Pg.Objects
         ' If the object is a TextBlock, set the font to Garamond,
         ' and set list entries at levels 2 through 5 to Italic.
         If obj.IsText Then
            obj.Font.FontName = "Garamond"
            For level% = 2 To 5
               obj.TextProperties(level%).Font.Italic = TRUE
            Next level%
         End If
      End ForAll
   End ForAll
End ForAll
```

The Application class Documents property returns an instance of the DocumentCollection class. Each element in the collection is a document, an instance of the Document class.

The Document class Pages property returns an instance of the PageCollection class. Each element in the collection is a page, an instance of the Page class.

The Page Objects property returns an instance of the ObjectCollection class. Some of the elements in this collection may be text blocks, instances of the TextBox class.

Format function

Formats a number, a date/time, or a string according to a supplied format.

Syntax

Format[**$**] **(** *expr* [**,** *fmt*] **)**

Elements

expr

Any expression. The expression is evaluated as a numeric expression if *fmt* is a numeric format, as a date/time if *fmt* is a date/time format, and as a string if *fmt* is a string format.

fmt

Optional. A format string: either a string consisting of the name of a format pre-defined in LotusScript, or else a string of format characters. If this format string is not supplied, Format[$] behaves like Str[$], omitting the leading space character for positive numbers.

Return value

Format returns a Variant containing a string, and Format$ returns a String.

If *expr* is a string and *fmt* is a numeric format string, LotusScript attempts to convert the string to a number. If successful, LotusScript then formats the result.

If the string can't be converted to a number, LotusScript attempts to interpret it as a date/time, and attempts to convert it to a numeric value. If successful, LotusScript then formats the result.

If *expr* can't be converted to the data type of the format string, Format returns *expr* without formatting it.

Formatting codes

Numeric formats

If *expr* is numeric, you can use one of the named numeric formats shown in the following section, or create a custom numeric format using the numeric formatting codes shown in the subsequent section.

Named numeric formats

Format name	*Display of the value of expr is ...*
General Number	As stored, without thousands separators.
Currency	As defined in the operating system's international settings. For example, you can format currency values with thousands separators, negative values in parentheses, and two digits to the right of the decimal separator. In OS/2, the function does not append the currency symbol to the number.
Fixed	With at least one digit to the left of the decimal separator, and with two digits to the right of the decimal separator.
Standard	With thousands separators, with at least one digit to the left of the decimal separator, and with two digits to the right of the decimal separator.
Percent	*expr* multiplied by 100, with at least one digit to the left of the decimal separator. Two digits are displayed to the right of the decimal separator, and a percent sign (%) follows the number.
Scientific	In standard scientific notation: with one digit to the left of the decimal separator and two digits to the right of the decimal separator, followed by the letter *E* or *e* and a number representing the exponent.
Yes/No	No if the number is 0, and Yes otherwise.
True/False	False if the number is 0, and True otherwise.
On/Off	Off if the number is 0, and On otherwise.

Custom numeric formatting codes

The following table describes the characters you can use in *fmt* to create custom formats for numeric values.

Formatting code	*Meaning*
"" (Empty string)	Display the number with no formatting
0 (zero)	Digit forced display. A digit is displayed for each zero in *fmt*, with leading or trailing zeros to fill unused spaces. All digits to the left of the decimal separator are displayed. If the number includes more decimal places than *fmt*, it is rounded appropriately.
# (pound sign)	Digit conditional display. The same display as 0 (digit forced display), except that no leading or trailing zeros are displayed.
. (period)	Decimal separator. The position of the decimal separator in *fmt*. Unless your formatting code includes a 0 immediately to the left of the decimal separator, numbers between -1 and 1 begin with the decimal separator. The actual decimal separator used in the returned formatted value is the decimal separator specified in the operating system's international settings.

continued

Formatting code	*Meaning*
% (percent sign)	Percentage placeholder. Multiplies the number by 100 and inserts the percent sign (%) in the position where it appears in *fmt*. If you include more than one percentage placeholder, the number is multiplied by 100 for each %. For example, %% means multiplication by 10000.
, (comma)	Thousands separator. To separate groups of three digits, counting left from the decimal separator, within numbers that include at least four digits to the left of the decimal separator, enclose the comma between a pair of the digit symbols 0 or #. The actual thousands separator used in the returned formatted value is the thousands separator specified in the operating system's international settings. A special case is when the comma is placed immediately to the left of the decimal separator (or the position of the implied decimal separator). This causes the number to be divided by 1000. For example, this returns "100": x = Format$(100000,"##0,.") If 100000 is replaced in this example by a number less than 1000 in absolute value, then this function returns "0."
E- E+ e- e+	Scientific notation. The number of digit symbols (0 or #) to the left of the decimal separator specifies how many digits are displayed to the left of the decimal separator, and the resulting magnitude of the exponent. Use E+ or e+ to display the sign of all exponents (the symbol + or -). Use E- or e- to display the sign of negative exponents only (the symbol -). All exponent digits are displayed, regardless of how many digit symbols follow the E-, E+, e-, or e+. If there are no digit symbols (the symbol 0 or #), an exponent of zero is not displayed; otherwise at least one exponent digit is displayed. Use 0 to format a minimum number of exponent digits, up to the maximum of three.
$ (dollar sign)	Currency symbol. Designates a currency value. The actual currency symbol used in the returned formatted value is the currency symbol specified in the operating system's international settings.
- + ( ) space	Literal characters. These are displayed as they appear in the format string.
\ (backslash)	Literal character prefix. The character following the backslash is displayed as is; for example, \# displays #. To display a backslash itself, precede it with another backslash; that is, \ \ displays \.
"ABC"	Literal string enclosed in double quotation marks. To specify the double quotation mark character in the *fmt* argument, you must use Chr(34). The characters enclosed in quotation marks are displayed as they appear in the format string.
; (semicolon)	Format section separator. Separates the positive, negative, zero, and NULL sections in *fmt*. If you omit the negative or zero format sections, but include the semicolons representing them, they are formatted like the positive section.

A custom format string for numeric values can have from one to four sections, separated by semicolons. In a format string with more than one section, each section applies to different values of *expr*. The number of sections determines the values to which each individual section applies. The following table describes how each section of a one-part or multi-part format string is used.

Number of sections	*Description*
One	The format applies to all numbers.
Two	The first section formats positive numbers and 0. The second section formats negative numbers.
Three	The first section formats positive numbers. The second section formats negative numbers. The third section formats 0.
Four	The first section formats positive numbers. The second section formats negative numbers. The third section formats 0. The fourth section formats NULL.

Date/time formats

Since date/time values are stored as floating point numbers, date/time values can be formatted with numeric formats. They can also be formatted with date/time formats. You can either use one of the named date/time formats shown in the following section, or create a custom date/time format using the date/time formatting codes shown in the subsequent section.

Named date/time formats

Format name	*Display of the date/time value is ...*
General Date	In a standard format. Converts a floating-point number to a date/time. If the number includes no fractional part, this displays only a date. If the number includes no integer part, this displays only a time.
Long Date	A Long Date as defined in the operating system's international settings.
Medium Date	dd-mmm-yy (yy/mmm/dd in Japan)
Short Date	A Short Date as defined in the operating system's international settings.
Long Time	A Long Time as defined in the operating system's international settings. Long Time always includes hours, minutes, and seconds.
Medium Time	Hours (0 - 12) and minutes using the time separator and AM/PM notation (AMPM notation in Japan)
Short Time	Hours (0 - 23) and minutes using only the time separator.

Custom date/time formatting codes

The following table describes the characters you can use in *fmt* to create custom formats for date/time values.

Formatting code	*Meaning*
: (colon)	Time separator. Separates hours, minutes, and seconds in formatted time values. The actual time separator used in the returned formatted value is the time separator specified for the given country in the operating system's international settings.
/ (slash)	Date separator. Separates day, month, and year in formatted date values. The actual date separator used in the returned formatted value is the date separator specified in the operating system's international settings.
c	Displays a date as ddddd, and a time as ttttt (see below). If the value includes no fractional part, only a date is displayed. If the value includes no integer part, only a time is displayed.
y	Day of the year as a number (1 - 366).
d	Day of the month as a number without a leading zero (1 - 31).
dd	Day of the month as a number with a leading zero (01 - 31).
ddd	Weekday as a three-letter abbreviation (Sun - Sat).
dddd	Weekday spelled out (Sunday - Saturday).
ddddd	Serial date number as a complete date (day, month, and year) formatted as an international Short Date string. If there is no Short Date string provided in the operating system, the date format defaults to mm/dd/yy.
dddddd	Serial date number as a complete date (day, month, and year) formatted as an international Long Date string. If there is no Long Date string provided in the operating system, the date format defaults to mmmm dd, yyyy.
w	Weekday as a number (1 - 7). Sunday is 1.
ww	Week of the year as a number (1 - 53).
m	Month of the year as a number without a leading zero (1 - 12). If the character is preceded by h in *fmt*, it displays the minute of the hour as a number without a leading zero (0 - 59).
mm	Month of the year as a number with a leading zero (01 - 12). If the character is preceded by h in *fmt*, it displays the minute of the hour as a number with a leading zero (00 - 59).
mmm	Month name as a 3-letter abbreviation (Jan - Dec).
mmmm	Month name spelled out (January - December).
q	Quarter of the year as a number (1 - 4).

continued

Formatting code	*Meaning*
yy	The last two digits of the year (00 - 99).
yyyy	The full (four-digit) year (0100 - 9999).
h	Hour of the day as a number without a leading zero (0 - 23).
hh	Hour of the day as a number with a leading zero (00 - 23).
n	Minute of the hour as a number without a leading zero (0 - 59).
nn	Minute of the hour as a number with a leading zero (00 - 59).
s	Second of the minute as a number without a leading zero (0 - 59).
ss	Second of the minute as a number with a leading zero (00 - 59).
ttttt	Time serial number as a complete time (including hour, minute, and second), formatted using the time separator provided in the operating system's international settings. A leading zero is displayed if the international leading zero setting is TRUE and the time is before 10:00 AM or PM. The default time format is h:mm:ss.
AM/PM am/pm	Uses hour values from 1 to 12, displaying AM or am for hours before noon, and PM or pm for hours after noon.
A/P a/p	Uses hour values from 1 to 12, displaying A or a for hours before noon, and P or p for hours after noon.
AMPM	Uses hour values from 1 to 12. Displays the contents of the 1159 string (s1159) in WIN.INI for hours before noon, and the contents of the 2359 string (s2359) for hours after noon. AMPM is case-insensitive, but the case of the string displayed matches the string as it exists in the operating system's international settings. The default format is AM/PM.

The following table shows some custom date/time formats applied to one date and time: 6:43:04 in the evening of April 12, 1995.

fmt	*Display*
m/d/yy	4/12/95
d-mmm-yy	12-Apr-95
d-mmmm	12-April
mmmm-yy	April-95
y	102
hh:mm AM/PM	06:43 PM
h:mm:ss a/p	6:43:04 p
h:mm	18:43
h:mm:ss	18:43:04
m/d/yy h:mm	4/12/95 18:43

String formatting codes

To format a string using Format or Format$, use the formatting codes in the following table to create a custom string format. There are no named string formats.

Custom string formats can have one section, or two sections separated by a semicolon (;). If the format has one section, the format applies to all strings. If the format has two sections, then the first applies to nonempty strings, and the second applies to the value NULL and the empty string ("").

The following table describes the characters you can use in *fmt* to create a custom string format.

Formatting code	*Meaning*
@ (at sign)	Character forced display. If the string being formatted includes a character in this position, display it. If not, display a space. @ is filled from right to left unless *fmt* contains an exclamation point (!).
& (ampersand)	Character optional display. If the string being formatted includes a character in this position, display it. If not, display nothing. & is filled from right to left unless *fmt* contains an exclamation point (!).
< (less-than sign)	All characters in the formatted string are displayed in lowercase.
> (greater-than sign)	All characters in the formatted string are displayed in uppercase.
! (exclamation point)	Forces @ and & to fill from left to right, rather than from right to left.

Formatting dates and times in Asian languages

The Format function supports additional formatting characters for dates and times in versions of LotusScript for Japan, People's Republic of China, Taiwan, and Korea.

Only single-byte characters are recognized as formatting characters. Double-byte characters are treated as literal characters. Some of the formatting characters for LotusScript in People's Republic of China and Taiwan are case-sensitive (see the following paragraphs); all of the other Asian language date/time formatting characters are case-insensitive.

When a date/time formatting code used in the Format function in LotusScript for Japan is also a date/time formatting code in WIN.INI, LotusScript for Japan interprets the code appropriately. For example, the formatting expression "Long Date" has the same meaning in LotusScript for Japan as in English-language LotusScript. (The meaning is to use the WIN.INI Long Date format.)

These formats only have meanings in Asian versions of Lotus products.

Date/time format codes

The first table shows the formatting codes for Japan.

Formatting code	*Meaning*
aaa	Weekday in abbreviated format (one double-byte character)
aaaa	Weekday in full format
e	Year in era (“0” suppressed)
ee	Year in era (“0” not suppressed)
g	Era name (single-byte one-character abbreviation)
gg	Era name (double-byte one-character abbreviation)
ggg	Full era name

This table shows the formatting codes for People’s Republic of China.

Formatting code	*Meaning*
aaaa	Weekday in full format (three double-byte characters)
O	Month (double-byte)
o	Month (single-byte)
A	Day (double-byte)
a	Day (single-byte)
E	Short year (double-byte)
e	Short year (single-byte)
EE	Long year (double-byte)
ee	Year (single-byte)

This table shows the formatting codes for Taiwan.

Formatting code	*Meaning*
aaaa	Weekday in full format (three double-byte characters)
O	Month (double-byte)
o	Month (single-byte)
A	Day (double-byte)
a	Day (single-byte)
E	Year in era (double-byte)
e	Year in era (single-byte)
EE	Year in era with era abbreviation (double-byte)
ee	Year in era with era abbreviation (single-byte)
EEE	Year in era with era name (double-byte)
eee	Year in era with era name (single-byte)
EEEE	Christian year with Christian era name (double-byte)
eeee	Christian year with Christian era name (single-byte)

This table shows the formatting codes for Korea.

Formatting code	*Meaning*
aaa	Weekday in abbreviated format (one double-byte character)
aaaa	Weekday in full format (three double-byte characters)

Examples: Format function

```
' Get monthly revenue and expenses from the user, converting strings
to
' currency. Compute and display the balance, formatted as currency.

Dim rev As Currency, expense As Currency, bal As Currency
rev@ = CCur(InputBox("How much did we make this month?"))
expense@ = CCur(InputBox("How much did we spend?"))
bal@ = rev@ - expense@
MessageBox "Our balance this month is " _
   & Format(bal@, "Currency")
```

Fraction function

Returns the fractional part of a number.

Syntax

Fraction (*numExpr* **)**

Elements

numExpr

Any numeric expression.

Return value

The data type of the return value is the same as the data type of *numExpr*.

Usage

The following table shows special cases of the return value:

numExpr	*Return value*
A date/time value	The time part
An integer	0
NULL	NULL

Examples: Fraction function

```
' Print the fractional part of PI
Print Fraction(PI)     ' Prints .141592653589793
```

FreeFile function

Returns an unused file number.

Syntax
FreeFile

Return value
FreeFile returns an Integer value.

Usage
Use FreeFile when you need a file number (to open a file), but you don't know what file numbers are currently available.

If no more file numbers are available, an error is generated.

LotusScript limits the number of open files to 255. Depending on your operating system environment and the Lotus product you are running, the actual number of files that you can open may be 15 or less. See your product documentation for details.

You can call the function as either FreeFile or FreeFile().

Examples: FreeFile function
```
Dim fileNum As Integer
Dim cdr As String
cdr$ = CurDrive() + "\AUTOEXEC.BAT"
' Assign the lowest available file number to fileNum.
fileNum% = FreeFile()
Print FreeFile()                        ' Prints 1 (1 is unused)
Open cdr$ For Input Access Read As fileNum%     ' Use file number 1
Print FreeFile()                        ' Prints 2 (1 is in use)
Close fileNum%
Print FreeFile()                        ' Prints 1 (1 is unused again)
```

Function statement

Defines a function.

Syntax
[**Static**] [**Public** | **Private**] **Function** *functionName* [**(** [*paramList*] **)**] [**As** *returnType*]

[*statements*]

End Function

Elements

Static

Optional. Specifies that the values of the function's local variables are saved between calls to the function.

Public | **Private**

Optional. Public specifies that the function is visible outside the scope (module or class) where the function is defined, as long as that remains loaded. Private specifies that the function is visible only within the current scope.

A function in module scope is Private by default; a function in class scope is Public by default.

functionName

The name of the function. This name can have a data type suffix character appended, to declare the type of the function's return value.

paramList

Optional. A comma-separated list of declarations indicating the parameters to be passed to this function in function calls.

The syntax for each parameter declaration is:

[**ByVal**] *parameter* [**( )** | **List**] [**As** *type*]

ByVal means that *parameter* is passed by value: that is, the value assigned to *parameter* is a local copy of a value in memory, rather than a pointer to that value.

*parameter***()** is an array variable. *parameter* **List** identifies *parameter* as a list variable. Otherwise, *parameter* can be a variable of any of the other data types that LotusScript supports.

As *dataType* specifies the variable's data type. You can omit this clause and append a data type suffix character to *parameter* to declare the variable as one of the scalar data types. If you omit this clause and *parameter* has no data type suffix character appended (and isn't covered by an existing Def*type* statement), its data type is Variant.

Enclose the entire list of parameter declarations in parentheses.

returnType

Optional. The data type of the value returned by the function.

returnType can be any of the scalar data types, or Variant, or a class name.

If **As** *returnType* is not specified, the function name's data type suffix character determines the return value's type. Do not specify both a *returnType* and a data type suffix character; LotusScript treats that as an error.

If you omit *returnType* and the function name has no data type suffix character appended, the function returns a value either of data type Variant or of the data type specified by a Def*type* statement.

Usage

The Public keyword cannot be used in a product object script or %Include file in a product object script, except to declare class members. You must put such Public declarations in (Globals).

Arrays, lists, type instances, and objects can't be passed by value as arguments. They must be passed by reference.

To return a value from a function, assign a value to *functionName* within the body of the function definition (see the example).

If you assign an array to *functionName*, you cannot refer to an element of *functionName* within the body of the function; such a reference will be taken as a recursive call of the function. To refer to an element of *functionName*, assign *functionName* to a variant variable and index the element there.

A function returns a value; a sub does not. To use the value returned by a function, put the function call anywhere in an expression where a value of the data type returned by the function is legal.

You don't have to use the value returned by a function defined by the Function statement. (The value returned by a built-in function must be used.) To call a function without using the return value, use the Call statement.

A function definition cannot contain another function or sub definition, or a property definition.

A function member of a class cannot be declared Static.

You can exit a function using an Exit Function statement.

Note If you're using a 32-bit version of Windows, an integer has four bytes; use the short integer (two bytes) to correspond to the LotusScript Integer when passing data to LotusScript.

Examples: Function statement

Use a sub and a function to compute the cost of buying a house as follows:

- Ask the user for the price of the house, and call the ComputeMortgageCosts sub with price as the argument.
- The ComputeMortgageCosts sub gathers down payment (at least 10% of cost), annual interest rate, and the term of the mortgage from the user, then calls the Payment function with 3 arguments. Annual interest and term (years) are passed by value rather than reference so the Payment function can adjust them to compute monthly rate and monthly payment without changing the values of these variables in the ComputeMortgageCosts sub.
- If the user enters positive values, Payment returns the monthly payment. Otherwise, it returns 0. ComputeMortgageCosts then constructs an appropriate message.

```
Dim price As Single, message As String

Function Payment (princpl As Single, _
                  ByVal intrst As Single, _
                  ByVal term As Integer) As Single
   intrst! = intrst! / 12
   term% = term% * 12
   ' If any of the arguments are invalid, exit the function
   ' (payment will return the value 0).
   If princpl! <= 0 Or intrst! <= 0 Or term% < 1 Then _
      Exit Function
   ' The standard formula for computing the amount of the
   ' periodic payment of a loan:
   Payment = princpl! * intrst! / (1 - (intrst! + 1) ^ (-term%))
End Function

Sub ComputeMortgageCosts (price As Single)
   Dim totalCost As Single, downpmt As Single
   Dim mortgage As Single, intrst As Single
   Dim monthlypmt As Single, years As Integer
EnterInfo:
   downpmt! = CSng(InputBox("How much is the down payment?"))
   ' The downpayment must be at least 10% of the price.
   If downpmt! < (0.1 * price!) Then
     MessageBox "Your down payment must be at least " _
     & Format(price! * .1, "Currency")
      GoTo EnterInfo
   Else
      mortgage! = price! - downpmt!
   End If
   intrst! = CSng(InputBox("What is the interest rate?"))
   years% = CInt(InputBox("How many years?"))
   ' Call the Payment function, which returns the
   ' monthly payment.
   monthlypmt! = Payment(mortgage!, intrst!, years%)
   totalCost! = downpmt! + (monthlypmt! * years% * 12)
   If monthlypmt! > 0 Then ' Create a multiline message.
      message$ = _
|Price | & Format(price!, "Currency") & |
Down Payment: | & Format(downpmt!, "Currency") & |
Mortgage: | & Format(mortgage!, "Currency") & |
Interest: | & Format(intrst!, "Percent") & |
Term: | & Str(years%) & | years
Monthly Payment: | & Format(monthlypmt!, "Currency") & |
Total Cost: | & Format(monthlypmt! * years% * 12, "Currency")
   Else
      message$ = "You did not enter valid input."
   End If
End Sub
```

```
' Start here.
price! = CSng(InputBox("How much does the house cost?"))
' Call the Compute MortgageCosts sub.
ComputeMortgageCosts (price!)
' Display the message.
MessageBox message$
```

Get statement

Reads data from a binary file or a random file into a variable.

Syntax

Get [#]*fileNumber* **,** [*recordNumber*] **,** *variableName*

Elements

fileNumber

The number assigned to the file when it was opened with the Open statement. Note that the pound sign (#), *fileNumber,* and *variableName* are all required.

recordNumber

Optional. The file position (the byte position in a binary file, or the record number in a random file) where data retrieval begins. If you omit *recordNumber*, LotusScript retrieves data beginning from the current file position.

variableName

The variable to be used for storing the retrieved data. *variableName* cannot be an array. However, a fixed-length array defined within a data type is allowed (this array could also contain other arrays as elements).

Usage

The first byte or record in a file is always file position 1. After each read operation, the file position is advanced:

- For a binary file, by the size of the variable
- For a random file, by the size of a record

The Get statement reads data into *variableName* depending on the variable's data type. The following table shows how the Get statement behaves for different data types.

variableName data type	*Get statement's behavior*
Variant	The Get statement interprets the first two bytes as the DataType of the data to be read. If the DataType is EMPTY or NULL, the Get statement stops reading data and sets *variableName* to EMPTY or NULL. If the DataType is numeric, the Get statement reads the appropriate number of bytes used to store data of that Data Type: Integer: 2 bytes Long: 4 bytes Single: 4 bytes Double: 8 bytes Currency: 8 bytes Date / time: 8 bytes
Fixed-length string	The Get statement reads the specified number of characters. For example, if a variable is declared as String*10, the Get statement reads exactly 10 characters.
Variable-length string	The Get statement behaves differently, depending on the type of file you're using. Random file: The first two bytes read indicate the string's length. The Get statement reads exactly that number of characters. If *variableName* is larger than a random file record, data is read from the file until *variableName* is filled. After *variableName* is filled, the file position is advanced to the next record. Binary file: The number of bytes read from the file is equal to the length of the string currently assigned to *variableName*. If *variableName* has not been initialized, no data is read from the file.
A variable of a user-defined type	The number of bytes required to read the data is the sum of the number of bytes required to read all members of the used-defined data type, which cannot contain a dynamic array, a list, or an object.

Examples: Get statement

```
Type PersonRecord
   empNumber As Integer
   empName As String * 20
End Type

Dim fileNum% As Integer
Dim fileName$ As String
Dim rec As PersonRecord
fileNum% = FreeFile()
fileName$ = "data.txt"
```

```
' Open a random file with record length equal to the
' size of the records in rec.
Open fileName$ For Random As fileNum% Len = Len(rec)

' Write a record at position 1.
rec.empNumber% = 123
rec.empName$ = "John Smith"
Put #fileNum%, 1, rec

' Write a record at position 2.
rec.empNumber% = 456
rec.empName$ = "Jane Doe"
Put #fileNum%, 2, rec

' Write a record at position 3.
rec.empNumber% = 789
rec.empName$ = "Jack Jones"
Put #fileNum%, , rec

' Rewind to the beginning of the file and print all records.
Seek fileNum%, 1
Do While Not EOF(fileNum%)
   Get #fileNum%, , rec
   Print rec.empNumber%; rec.empName$
   ' The Get function advances to the next record automatically.
Loop

Close fileNum%

' Prints three records:
'  123 John Smith
'  456 Jane Doe
'  789 Jack Jones
```

GetFileAttr function

Retrieves file-system attributes of a file or directory.

Syntax

GetFileAttr (*fileName* **)**

GetAttr is acceptable in place of GetFileAttr.

Elements

fileName

The name of a file or directory. File and directory names can optionally include paths.

Return value

The return value is the sum of the Integer values in the following list for those attributes that apply to *fileName*:

Value	*Attribute*	*Constant*
0	Normal file	ATTR_NORMAL
1	Read-only file	ATTR_READONLY
2	Hidden file	ATTR_HIDDEN
4	System file	ATTR_SYSTEM
16	Directory	ATTR_DIRECTORY
32	File that has changed since it was last backed up (archived)	ATTR_ARCHIVE

Usage

The constants in the preceding list are defined in the file lsconst.lss. Including this file in your script allows you to use constant names instead of their numeric values.

Examples: GetFileAttr function

```
' This example creates a file, calls SetFileAttr to set its attributes
' to Read-Only, System, and Hidden, and then calls GetFileAttr to
' determine the file attributes.

%Include "lsconst.lss"

Dim fileNum As Integer, attr As Integer
Dim fileName As String, msg As String
fileNum% = FreeFile()
fileName$ = "data.txt"

Open fileName$ For Output As fileNum%
Close fileNum%
SetFileAttr fileName$, ATTR_READONLY + ATTR_SYSTEM + ATTR_HIDDEN
attr% = GetFileAttr(fileName$)
If (attr% And ATTR_READONLY) Then
  msg$ = msg$ & " Read-Only "
Else
  msg$ = msg$ & " Normal "
End If
If (attr% And ATTR_HIDDEN)    Then msg$ = msg$ & " Hidden "
If (attr% And ATTR_SYSTEM)    Then msg$ = msg$ & " System "
If (attr% And ATTR_DIRECTORY) Then msg$ = msg$ & " Directory "
Print msg$

SetFileAttr fileName$, ATTR_NORMAL  ' Reset to normal.
Kill fileName$
```

GetObject function

Opens an OLE Automation object contained in an application file, or returns the currently active OLE Automation object of the specified class.

Note GetObject is not supported under OS/2, under UNIX, or on the Macintosh.

Syntax

GetObject (*pathName* [**,** *className*] **)**

Elements

pathName

Either a string containing the full path and file name of an application file or an empty string. The application must support OLE Automation. If *pathName* is the empty string (""), you must specify a *className.*

className

A string of the form *appName.appClass* that identifies the application in which the class is defined and the class of the object to retrieve (for example, "WordPro.Application").

appName is the name of an application that supports OLE Automation. *appClass* is the name of the class of which you want to retrieve an instance.

Return value

GetObject returns an OLE Automation object reference.

Usage

Use the Set statement to assign the object reference returned by GetObject to a Variant variable.

If the application specified by *appName* is not already running, GetObject starts it before retrieving the OLE Automation object. References to the object remain valid only while the application is running. If the application terminates while you are using the object reference, LotusScript generates a run-time error.

If *pathName* is the empty string (""), GetObject retrieves the currently active object of the specified class. If no object of that class is active, an error occurs.

If *className* is omitted, GetObject determines the application to run and the object to retrieve based on the *pathName.* This form of GetObject is useful only when the application file contains a single object.

Each product that supports OLE Automation provides one or more classes. See the product's documentation for details.

LotusScript supports the following return types for OLE properties and methods. Only an OLE method or property can return a type designated as "OLE only."

OLE return type	*Description*
VT_EMPTY	(No data)
VT_NULL	(No data)
VT_I2	2-byte signed integer
VT_I4	4-byte signed integer
VT_R4	4-byte real
VT_R8	8-byte real
VT_CY	Currency
VT_DATE	Date
VT_BSTR	String
VT_DISPATCH	IDispatch, OLE only
VT_ERROR	Error, OLE only
VT_BOOL	Boolean
VT_VARIANT	(A reference to data of any other type)
VT_UNKNOWN	IUnknown, OLE only
VT_ARRAY	(An array of data of any other type)

You can use a ForAll loop to iterate over the members of OLE collections.

LotusScript supports passing arguments to OLE properties. For example:

```
' Set v.prop to 4; v.prop takes two arguments.
v.prop(arg1, arg2) = 4
```

LotusScript does not support identifying arguments for OLE methods or properties by name rather than by the order in which they appear, nor does LotusScript support using an OLE name by itself (without an explicit property) to identify a default property.

Results are unspecified for arguments to OLE methods and properties of type Boolean, byte, and date that are passed by reference. LotusScript does not support these data types.

The word GetObject is not a LotusScript keyword.

Examples: GetObject function

```
Dim myDoc As Variant

' Get the WordPro.Application object from a file.
Set myDoc = GetObject("c:\status.sam", "WordPro.Application")

' Call the Print method defined for WordPro.Application object.
myDoc.Print
```

GoSub statement

Transfers control in a procedure to a labeled statement, with an optional return of control.

Syntax

GoSub *label*

Elements

label

The label of a statement to which you want to transfer control.

Usage

You can't use the GoSub statement at the module level; you can only use it in a procedure. The GoSub statement, its label, and the Return statement must all reside in the same procedure.

When LotusScript encounters a GoSub statement, execution branches to the specified labeled statement and continues until either of two things happen:

- LotusScript encounters a Return statement, at which point execution continues from the statement immediately following the GoSub statement.
- LotusScript encounters a statement such as Exit or GoTo, which passes control to some other part of the script.

Examples: GoSub statement

```
' In response to user input, LotusScript transfers control to one of
' three labels, constructing an appropriate message, and continues
' execution at the statement following the GoSub statement.

Sub GetName
   Dim yourName As String, Message As String
   yourName$ = InputBox$("What is your name?")
   If yourName$ = "" Then    ' The user enters nothing.
      GoSub EmptyString
   ' Do a case-insensitive comparison.
   ElseIf LCase(yourName$) = "john doe" Then
      GoSub JohnDoe
   Else
      Message$ = "Thanks, " & yourName$ _
         & ", for letting us know who you are."
   End If
   ' The Return statements return control to the next line.
   MessageBox Message$
   Exit Sub

EmptyString:
   yourName$ = "John Doe"
```

```
   Message$ = "Okay! As far as we're concerned, " _
      & "your name is " & yourName$ & ", and you're on the run!"
   Return

JohnDoe:
   Message$ = "We're on your trail, " & yourName$ _
      & ". We know you are wanted dead or alive!"
   Return
End Sub
GetName    ' Call the GetName sub.
```

GoTo statement

Transfers control within a procedure to a labeled statement.

Syntax

GoTo *label*

Elements

label

A label of a statement to which you want to transfer control.

Usage

You can't use the GoTo statement at the module level; you can only use it in a procedure. You can't use GoTo to transfer control into or out of a procedure or a With...End With block.

Use the GoTo statement to transfer control to any labeled statement that does not violate either of the preceding rules.

Examples: GoTo statement

This example illustrates On Error...GoTo, On...GoTo, Resume...GoTo, and GoTo.

The user enters a value. If the value is 1, 2, or 3, the On...GoTo statement transfers control to label1, label2, or label3. If the value is another number in range for On...GoTo (the range is 0-255), control moves on the next statement. If the user enters a number that is out of range for On...GoTo or that the CInt function cannot convert to an integer, an error condition occurs, and the OnError...GoTo statement transfers control to the OutOfRange label.

Depending on the user's entry, the OneTwoThree sub displays an appropriate message. If the entry is valid, an Exit Sub statement exits the Sub. If the entry is not valid, a GoTo statement transfers control to the EnterNum label, and the user is given another chance to make a valid entry.

```
Sub OneTwoThree
   Dim num As Integer
   On Error GoTo OutOfRange
EnterNum:
   num% = CInt(InputBox("Enter 1, 2, or 3"))
   On num% GoTo label1, label2, label3
  ' The user did not enter 1, 2, or 3, but a run-time error
  ' did not occur (the user entered a number in the range 0-255).
   MessageBox "You did not enter a correct value! Try again!"
   GoTo EnterNum
label1:
   MessageBox "You entered 1."
   Exit Sub
label2:
   MessageBox "You entered 2."
   Exit Sub
label3:
   MessageBox "You entered 3."
   Exit Sub
   ' An error condition has occurred.
OutOfRange:
   MessageBox "The value you entered is negative, " _
      & "greater than 255, or is not a number. Try again!"
   GoTo EnterNum
End Sub
OneTwoThree   ' Call the OneTwoThree sub.
```

Hex function

Return the hexadecimal representation of a number as a string.

Syntax

Hex[**$**] **(** *numExpr* **)**

Elements

numExpr

Any numeric expression. If *numExpr* evaluates to a number with a fractional part, LotusScript rounds it to the nearest integer before deriving its hexadecimal representation.

Return value

Hex returns a Variant of DataType 8 (String), and Hex$ returns a String.

Return values will only include the characters 0 - 9 and A - F, inclusive. The maximum length of the return value is eight characters.

Usage

If the data type of *numExpr* is not Integer or Long, LotusScript attempts to convert it to a Long. If it cannot be converted, an error occurs.

Examples: Hex function

```
Print Hex$(15)              ' Prints "F"

' Converts Double argument to Long.
Print Hex$(15.0)            ' Prints "F"

' Rounds Double argument, then converts to Long.
Print Hex$(15.3)            ' Prints "F"

' Computes product 14.841, rounds to 15.0, then converts to 15.
Print Hex$(15.3 * .97)                ' Prints "F"
```

Hour function

Returns the hour of the day for a date/time argument as an integer from 0 to 23.

Syntax

Hour (*dateExpr* **)**

Elements

dateExpr

Any of the following:

- A valid date/time string of String or Variant data type. LotusScript interprets a 2-digit year designation in a date/time string as that year in the twentieth century. For example, 17 and 1917 are considered the same.
- A Variant with a value of DataType 7 (Date/Time)
- A number within the valid date range: the range -657434 (Jan 1, 100 AD) to 2958465 (Dec 31, 9999 AD), inclusive
- NULL

Return value

Hour returns a Variant containing a value of DataType 2 (Integer). If the *dateExpr* is a Variant containing the value NULL, then Hour returns NULL.

Examples: Hour function

```
' Construct a message that displays the current time and
' the number of hours, minutes, and seconds remaining in the day.
Dim timeFrag As String, hoursFrag As String
Dim minutesFrag As String, secondsFrag As String
Dim crlf As String, message As String
timeFrag$ = Format(Time, "h:mm:ss AM/PM")
hoursFrag$ = Str(23 - Hour(Time))
minutesFrag$ = Str(59 - Minute(Time))
secondsFrag$ = Str(60 - Second(Time))
crlf$ = Chr(13) & Chr(10)              ' Carriage return/line feed
message$ = "Current time: " & timeFrag$ & ". " & crlf$ _
   & "Time remaining in the day: " _
   & hoursFrag$ & " hours, " _
   & minutesFrag$ & " minutes, and " _
   & secondsFrag$ & " seconds."
MessageBox(message$)
```

If...GoTo statement

Conditionally executes one or more statements or transfers control to a labeled statement, depending on the value of an expression.

Syntax

If *condition* **GoTo** *label* [**Else** [*statements*]]

Elements

condition

Any numeric expression. A value of 0 is interpreted as FALSE, and any other value is interpreted as TRUE.

label

The name of a label.

statements

A series of statements, separated by colons.

Usage

An If...GoTo statement must occupy a single line of code — line continuation with the underscore character (_) is allowed.

If *condition* is TRUE, LotusScript executes the GoTo statement, transferring control to the statement following the label *label*. If *condition* is FALSE, LotusScript executes the block of *statements* in the Else clause. If there is no Else clause, execution continues with the next statement.

You can't use an If...GoTo statement to transfer control into or out of a procedure, and you can't use it at the module level.

Examples: If...GoTo statement

Ask the user to propose a down payment for a house. Elsewhere, the cost has been set at $235,000. Depending on whether or not the user proposes a down payment of at least 10% of cost, respond accordingly.

```
Sub ProcessMortgage(cost As Single)
   Dim downpmt As Single, msg As String
   msg$ = "Cost: " + Format(cost!, "Currency") _
         & ". Enter a down payment:"
   downpmt! = CSng(InputBox(msg$))
   If downpmt! < .1 * cost! GoTo NotEnough
   msg$ = Format(downpmt!, "Currency") & " will do fine!"
   MessageBox msg$
   ' Continue processing the application
   ' ...
   ' ...
   Exit Sub

   NotEnough:
   msg$ = "Sorry, " & Format(downpmt!, "Currency") _
       & " is not enough!"
   MessageBox msg$
End Sub

Dim cost As Single
cost! = 235000
ProcessMortgage(cost!)    ' Call the ProcessMortgage sub.
```

If...Then...Else statement

Conditionally executes one or more statements, depending on the value of an expression.

Syntax

If *condition* **Then** [*statements*] [**Else** [*statements*]]

Elements

condition

Any numeric expression. A value of 0 is interpreted as FALSE, and any other value is interpreted as TRUE.

statements

A series of statements, separated by colons.

Usage

An If...Then...Else statement must occupy a single line of code—line continuation with the underscore character (_) is allowed.

If *condition* is TRUE, the statements following Then, if any, are executed. If *condition* is FALSE, the statements following Else are executed.

If no statements follow Then, and there is no Else clause, Then must be followed by a colon (:). Otherwise LotusScript assumes that the statement is the first line of an If...Then...Else...End If statement.

Examples: If...Then...Else statement

```
Dim x As Integer
If x% > 0 Then Print FALSE Else Print TRUE
' Output:
' True
```

The initial value of x is 0, so LotusScript prints True.

If...Then...ElseIf statement

Conditionally executes a block of statements, depending on the value of one or more expressions.

Syntax

If *condition* **Then**

statements

[**ElseIf** *condition* **Then**

statements]

...

[**Else**

statements]

End If

Elements

condition

Any numeric expression. A value of 0 is interpreted as FALSE, and any other value is interpreted as TRUE.

statements

Statements that are executed if *condition* is TRUE.

Usage

LotusScript executes the statements following the Then keyword for the first condition whose value is TRUE. It evaluates an ElseIf condition if the preceding condition is FALSE. If none of the conditions is TRUE, LotusScript executes the statements following Else keyword. Execution continues with the first statement following the End If statement.

You can include any number of ElseIf expressions in the block.

You can include an If statement within an If statement. Each If block must be terminated by an End If.

Examples: If...Then...ElseIf statement

```
Dim quantity As Integer, pctDiscount As Single
Dim unitPrice As Currency, total As Currency
unitPrice@ = 3.69
quantity% = 50

' Define discount based on quantity purchased.
If quantity% > 99 Then
   pctDiscount! = .20
ElseIf quantity% > 49 Then
   pctDiscount! = .10
Else
   pctDiscount! = 0
End If
total = (quantity% * unitPrice@) * (1 - pctDiscount!)
Print   "Unit price: $"; unitPrice@, _
  "Quantity: "; quantity%, _
  "Discount%: "; pctDiscount!, _
  "Total: $"; total@
```

%If directive

Conditionally compiles a block of statements, depending on the value of one or more product constants.

Syntax

%If *productConst*

statements

[**%ElseIf** *productConst*

statements]

...

[**%Else**

statements]

%End If

Elements

productConst

A constant defined by a Lotus product, or one of the platform-identification constants described below. Refer to the product's documentation for a list of product-defined constants.

statements

Statements that are compiled if *productConst* evaluates to TRUE.

Usage

You cannot enter %If, %ElseIf, %Else, and %End If directly in the IDE. You must enter these directives in a file and insert the file in the IDE with the %Include directive.

productConst must appear on the same line as %If or %ElseIf. Nothing except a comment can appear on the same line following %If *productConst* or %ElseIf *productConst*, or on the same line with %Else or %End If. None of these lines can be continued with the underscore character (_).

To test each %If condition or %ElseIf condition in this statement, the LotusScript compiler calls the Lotus product to evaluate the constant *productConst*. The product returns either TRUE (-1) or FALSE (0).

A condition is evaluated only if the product returns FALSE for the preceding condition. LotusScript compiles the *statements* for the first %If condition or %ElseIf condition that the product evaluates as TRUE. Once this happenss, no further conditions are evaluated, and no further *statements* are compiled.

If neither the %If condition nor any %ElseIf condition evaluates to TRUE, the %Else *statements* (if any) are compiled.

You can include any number of %ElseIf directives in the block.

You can't include an %If block within an %If block.

LotusScript implements the constants in the following table as product #defines. When one of these is used as *productConst*, the LotusScript compiler does not call the product to evaluate *productConst*. LotusScript itself evaluates the constant as TRUE or FALSE. The value of each constant depends on the platform LotusScript is running on.

Constant	*Platform or functionality*
WIN16	Windows with 16-bit API (Windows 3.1)
WIN32	Windows with 32-bit API (Windows NT or Windows 95)
WINNT	Windows NT
WIN95	Windows 95
WIN40	Windows 95 or Windows NT 4.0
WINDOWS	Any Windows platform type (any of the above WIN*xx* constants)
HPUX	HP® / UNIX 9.X or greater
SOLARIS	SunOS™ 5.0 or greater
UNIX	Any UNIX type (HP_UX® or Solaris®)
OS2	OS / 2, version 2.0 or greater
MAC	Macintosh System 7™
OLE	OLE-2 is available
MAC68K	Macintosh Motorola® 68000 version (running on either a 68*xxx* Macintosh or the PowerPC™)
MACPPC	Macintosh PowerPC version

For example, here are several platforms and the constants that identify them:

Windows 3.1
    WIN16, WINDOWS

Windows 95
    WIN32, WIN95, WIN40, WINDOWS

HP / UNIX 9.X
    HPUX, UNIX

Examples: %If directive

This example compiles and runs in either Windows 3.1, Windows NT, or Windows 95. Depending on whether the application is compiled and run under 16-bit Windows (Windows 3.1) or 32-bit Windows (Windows 95 or Windows NT), you should declare and use an appropriate Windows handle variable and the appropriate version of two Windows API functions.

GetActiveWindow returns the handle (an Integer in 16-bit Windows, a Long in 32-bit Windows) of the currently active window. GetWindowText returns the text in the window title bar.

```
Dim winTitle As String * 80
%If WIN16                              ' 16-bit Windows
   Dim activeWin As Integer            ' Window handles are Integer.
   Declare Function GetActiveWindow% Lib "User" ()
   Declare Function GetWindowText% Lib "User" _
         (ByVal hWnd%, ByVal lpstr$, ByVal i%)
%ElseIf WIN32                          ' 32-bit Windows
   Dim activeWin As Long               ' Window handles are Long.
   Declare Function GetActiveWindow& Lib "User32" ()
   Declare Function GetWindowText% Lib "User32" _
          Alias "GetWindowTextA" _
          (ByVal hWnd&, ByVal lpstr$, ByVal i&)
%End If

' Print the name of the currently active window.
activeWin = GetActiveWindow()          ' Returns an Integer or a Long.
Call GetWindowText(ActiveWin, winTitle$, 80)
Print winTitle$
```

IMEStatus function

Returns an integer indicating the current input mode (IME) for extended character sets.

Syntax

IMEStatus

Return value

IMEStatus provides support for languages that use extended character sets. The function returns a status code indicating the current input mode (IME). This code depends on the country for which the Lotus product is built. The following table describes the return values. For countries not listed in the table, the return value is 0.

Country	*Constant*	*Value*	*Description*
All	IME_NOT_INSTALLED	0	IME is not installed
	IME_ON	1	IME is on
	IME_OFF	2	IME is off
Japan	IME_HIRAGANA	4	Double-byte Hiragana
	IME_KATAKANA_DBCS	5	Double-byte Katakana
	IME_KATAKANA_SBCS	6	Single-byte Katakana
	IME_ALPHA_DBCS	7	Double-byte alphanumeric
	IME_ALPHA_SBCS	8	Single-byte alphanumeric

continued

Country	*Constant*	*Value*	*Description*
Taiwan	IME_NATIVE_MODE	4	Taiwan native mode
	IME_ALPHA_DBCS	7	Double-byte alphanumeric
	IME_ALPHA_SBCS	8	Single-byte alphanumeric
Korea	IME_HANGEUL	4	Hangeul DBC
	IME_HANJACONVERT	5	Hanja conversion
	IME_ALPLHA_DBCS	7	Double-byte alphanumeric
	IME_ALPHA_SBCS	8	Single-byte alphanumeric
PRC	IME_NATIVE_MODE	4	PRC native mode
	IME_ALPHA_DBCS	7	Double-byte alphanumeric
	IME_ALPHA_SBCS	8	Single-byte alphanumeric

Usage

IMEStatus is supported for Windows DBCS only. The Taiwan and PRC codes are supported for Win95 only.

%Include directive

At compile time, inserts the contents of an ASCII file into the module where the directive appears.

Syntax

%Include *fileName*

Elements

fileName

A string literal whose value is a file name; you can optionally include a path.

If you omit the file name extension, LotusScript assumes .lss. To include a file that has no extension, include a period at the end of the the file name. For example:

```
%Include "orfile."
```

This prevents LotusScript from adding the .lss extension to the file name.

Usage

The %Include directive must be the only item on a line, except for an optional trailing comment. It must be followed by white space (a space character, a tab character, or a newline character).

If you don't specify a path for the included file, the search path depends on the specific Lotus product you're using.

An included file can itself contain %Include directives. You can nest up to 16 files.

At compile time, LotusScript replaces the %Include directive with the entire contents of the named file. They are then compiled as part of the current script.

If a run-time error occurs in a statement in an included file, the line number reported is that of the %Include directive.

If a compile-time error occurs in a statement in an included file, the file name and the line number within that included file are reported with the error.

The file you include must be a text file containing only LotusScript statements. If anything in the included file cannot be compiled, LotusScript generates a compiler error.

If the file is not found, LotusScript generates an error.

Examples: %Include directive

```
' Include the contents of c:\testfile.txt with
' the current script when it is compiled.
%Include "c:\testfile.txt"
```

Input # statement

Reads data from a sequential file and assigns that data to variables.

Syntax

Input #*fileNumber* **,** *variableList*

Elements

fileNumber

The number assigned to the file when you opened it. A pound sign (#) sign must precede the file number.

variableList

A comma-separated list of variables. The data read from the file is assigned to these variables. File data and its data types must match these variables and their data types.

variableList cannot include arrays, lists, variables of a user-defined data type, or object reference variables. It can include individual array elements, list elements, and members of a user-defined data type or user-defined class.

Usage

The following table shows how the Input # statement reads characters for various data types.

variableList data type	*How Input # reads characters*
Numeric variable	The next non-space character in the file is assumed to begin a number. The next space, comma, or end-of-line character in the file ends the number. Blank lines and non-numeric values are translated to the number 0.
String variable	The next non-space character in the file is assumed to begin a string. Note these special conditions: If that character is a double quotation mark (“), it is ignored; however, all characters following it (including commas, spaces, and newline characters) up to the next double quotation mark are read into the string variable. If the first character is not a double quotation mark, the next space, comma, or end-of-line character ends the string. Blank lines are translated to the empty string (”“). Note that tab is a non-space character.
Fixed-length string variable	LotusScript reads this according to its length. For example, LotusScript reads a variable declared as String *10 as 10 bytes.
Variant variable	The next non-space character in the file is assumed to begin the data. If the data is: Empty (a delimiting comma or blank line), LotusScript assigns the variable the EMPTY value. The literal ”#NULL#“, LotusScript assigns the variable the NULL value. A date/time literal, LotusScript assigns the variable the DataType 7 (Date/Time). A whole number, LotusScript assigns the variable the Data Type 2 (integer) if the number is in the legal range for integer; the DataType 3 (Long) if the number is in the legal range for Long but not within the range for integer; and otherwise the DataType 5 (Double). A number with a fractional part, LotusScript assigns the variable the DataType 5 (Double). If none of the above applies, LotusScript assigns the variable the String type.

If LotusScript encounters an EOF (end-of-file), input terminates and an error is generated.

LotusScript inserts a “\n” character in any multi-line string (for example, a string that you type in using vertical bars or braces). If you Print the string to a file, the \n will be interpreted as a newline on all platforms. If you Write the string to a file, the \n may not be interpreted as a newline on all platforms. Therefore, when reading a multi-line string from a sequential file, use Input, not Line Input.

When reading record-oriented data, using a random file with the Get statement is easier and more efficient than using Input #.

Examples: Input # statement

```
Dim fileNum As Integer, empNumber As Integer, i As Integer
Dim fileName As String, empName As String
Dim empLocation As Variant
Dim empSalary As Currency

fileNum% = FreeFile()
fileName$ = "data.txt"

' Write out some employee data.
Open fileName$ For Output As fileNum%
Write #fileNum%, "Joe Smith", 123, "1 Smith Road", 25000.99
Write #fileNum%, "Jane Doe", 456, "Two Cambridge Center", 98525.66
Write #fileNum%, "Jack Jones", 789, "Fourth Floor", 0
Close fileNum%

' Now read it all back and print it.
Open fileName$ For Input As fileNum%
For i% = 1 To 3
   Input #fileNum%, empName$, empNumber%, empLocation, empSalary@
   Print empName$, empNumber%, empLocation, empSalary@
Next i%

' Output:
' Outputs the following groups of four values, each
' consisting of String, Integer, Variant, and Currency values.

' Joe Smith     123          1 Smith Road                25000.99
' Jane Doe      456          Two Cambridge Center        98525.66
' Jack Jones    789          Fourth Floor                0

Close fileNum%
```

Input function

Reads a sequence of characters from a sequential or binary file into a string variable, without interpreting the input.

Syntax

Input[**$**] **(** *count* **,** [**#**]*fileNumber* **)**

Elements

count

The number of characters to read. *count* must not exceed 32000.

fileNumber
The number assigned to the file when you opened it.

Return value

The Input function returns a Variant, and Input$ returns a String.

LotusScript returns the specified number of characters, beginning at the current position in the file.

If you request more characters than are available, then the available characters are returned in the string, the length of the returned string is less than *count*, and LotusScript generates an error.

If *count* is 0, LotusScript returns the empty string (“”).

Usage

The input data is not filtered or translated in any way. All characters are returned, including newline characters, quotation marks, and spaces.

If you want to work with bytes instead of characters, use the InputB or InputB$ function.

You cannot use the Input, Input$, InputB, or InputB$ functions to read a file opened in Output, Append, or Random mode.

Examples: Input function

```
Dim fileNum As Integer
Dim fileName As String
Dim firstCheck As String

fileNum% = FreeFile()
fileName$ = "data.txt"

' Write out some employee data.
Open fileName$ For Output As fileNum%
Write #fileNum%, "Joe Smith", 123, "1 Smith Road", 25000.99
Write #fileNum%, "Jane Doe", 456, "Two Cambridge Center", 98525.66
Close fileNum%

' Read in first 23 characters of data and print.
Open fileName$ For Input As fileNum%
firstCheck$ = Input$(23, fileNum%)
Print firstCheck$   ' Output:  "Joe Smith",123,"1 Smit
Close fileNum%
```

InputB function

Reads a sequence of bytes from a sequential or binary file into a string variable without interpreting the input.

Syntax

InputB[**$**] **(** *count* **,** [**#**]*fileNumber* **)**

count
The number of bytes to read. The *count* must not exceed 64000 (this number of bytes represents 32000 characters).

fileNumber
The number assigned to the file when it was opened.

Return value

The InputB function returns a Variant, and InputB$ returns a String.

LotusScript returns the specified number of bytes, beginning at the current position within the file. If you request more bytes than are available, then the available bytes are returned in the string and LotusScript generates an error.

The length of the returned string (measured in characters, as computed by the Len function) is (*# bytes returned*) / 2 if an even number of bytes is returned, and otherwise (*# bytes returned* + 1) / 2, if an odd number of bytes is returned. If an odd number of bytes is returned, then the last character in the returned string is padded with a 0 byte.

If *count* is 0, LotusScript returns the empty string (“”).

Usage

The input data is not filtered or translated in any way. All bytes are returned, including the bytes representing newline, quotation marks, and space.

If you want to work with characters instead of bytes, use the Input or Input$ function.

You cannot use the Input, Input$, InputB, or InputB$ function to read a file opened in Output, Append, or Random mode.

Examples: InputB function

```
Print InputB$(4, 1)   ' Prints the next four bytes from file number 1.
```

InputBox function

Displays a dialog box containing a prompt for user entry, and returns input from the user as a string.

Syntax

InputBox[**$**] **(** *prompt* [**,** [*title*] [**,** [*default*] [**,** *xpos* **,** *ypos*]]] **)**

Elements

prompt

A string expression. This is the message displayed in the dialog box. *prompt* can be up to 128 characters in length.

title

Optional. A string expression. This is displayed in the title bar of the dialog box. *title* can be up to 128 characters in length.

If you omit *title*, nothing is displayed in the title bar. If you omit *title* and specify either *default* or *xpos* and *ypos*, include a comma in place of *title*.

default

Optional. A string expression. This is displayed in the text entry field in the dialog box as the default user response. *default* can be up to 512 characters in length.

If you omit *default*, the text input box is empty. If you omit *default* and specify *xpos* and *ypos*, include a comma in place of *default*.

xpos

Optional. A numeric expression that specifies the horizontal distance, in units of 1 pixel, between the left edge of the dialog box and the left edge of the display screen. If you omit *xpos*, the distance is 0. If you specify *xpos*, you have to specify *ypos* as well.

ypos

Optional. A numeric expression that specifies the vertical distance, in units of 1 pixel, between the top edge of the dialog box and the top edge of the screen. If you omit *ypos*, the distance is 0. If you specify *ypos*, you have to specify *xpos* as well.

Return value

The InputBox function returns a Variant containing a string. InputBox$ returns a String.

Usage

InputBox displays a dialog box with OK and Cancel buttons and a text entry field, interrupting execution of the script until the user confirms the text entry by clicking OK or Cancel. Then InputBox returns that entry. If the user clicks Cancel, InputBox returns the empty string (“”). When the user clicks OK or Cancel, execution resumes.

The Lotus product where you are running LotusScript may allow longer strings than described above for *prompt*, *title*, *default*, and the text entered into the text entry field.

LotusScript will support longer strings for these items if the Lotus product does, up to 16000 characters.

Examples: InputBox function

```
' Ask the user for an integer. Convert user input
' from a string to an integer.

Dim num As Integer
num% = CInt(InputBox$("How many do you want?"))
```

InputBP function

Reads a sequence of bytes (in the platform-native character set) from a sequential or binary file into a string variable without interpreting the input.

Syntax

InputBP[**$**] **(** *count* **,** [**#**]*fileNumber* **)**

count
The number of bytes to read. The *count* must not exceed 32000 (this number of bytes represents 32000 ASCII characters and 16000 DBCS characters).

fileNumber
The number assigned to the file when it was opened.

Return value

The InputBP function returns a Variant, and InputBP$ returns a String.

LotusScript returns the specified number of bytes, beginning at the current position within the file. If you request more bytes than are available, then the available bytes are returned in the string and LotusScript generates an error.

The length of the returned string (measured in characters, as computed by the Len function) is the number of Unicode characters that the bytes translate into. For example, 10 bytes of ASCII characters translate into 10 Unicode characters; 10 bytes of DBCS characters translate into 5 Unicode characters. If the last requested byte read is the lead byte of a DBCS character, the byte is dropped and the file pointer is positioned one byte before the last requested byte.

If *count* is 0, LotusScript returns the empty string (“”).

Usage

The input data is translated into Unicode.

If you want to work with characters instead of platform bytes, use the Input or Input$ function. If you want to work with untranslated bytes, use the InputB or InputB$ function.

You cannot use the Input, Input$, InputB, InputB$, InputBP, or InputBP$ function to read a file opened in Output, Append, or Random mode.

Examples: InputBP function

```
Print InputBP(4, 1)   ' Prints the next four bytes from file number 1.
```

InStr function

Returns the position of the character that begins the first occurrence of one string within another string.

Syntax

InStr ([*begin* **,**] *string1* **,** *string2* [**,** *compMethod*] **)**

Elements

begin

Optional. A numeric expression with a positive integer value. *begin* specifies the character position in *string1* where InStr should begin searching for *string2*. If you omit *begin*, it defaults to 1. If you specify *compMethod*, you must specify *begin* as well.

string1

The string that InStr searches for the occurrence of *string2*.

string2

The string for which InStr searches to see if it occurs in *string1*.

compMethod

A number designating the comparison method: 0 for case sensitive and pitch sensitive, 1 for case insensitive and pitch sensitive, 4 for case sensitive and pitch insensitive, 5 for case insensitive and pitch insensitive. If you omit *compMethod*, the default comparison mode is the mode set by the Option Compare statement for this module. If there is no statement for the module, the default is case sensitive and pitch sensitive.

Return value

InStr returns the character position of the first occurrence of *string2* within *string1*. The following table shows how the function responds to various conditions.

Condition	*Return value*
string1 is the empty string ("")	0
string2 is not found after *begin* in *string1*	0
begin is larger than the length of *string1*	0
string2 is the empty string ("")	The value of *begin*. If you omit *begin*, InStr returns the value 1.
string1 is NULL	NULL
string2 is NULL	NULL
begin or *compMethod* is NULL	Error

Usage

If you want to work with bytes, use the InStrB function.

Examples: InStr function

```
' The value 5 (the position of the character where the first
' occurrence of LittleString begins in BigString) is assigned
' to the variable positionOfChar.

Dim big As String, little As String
Dim positionOfChar As Long
big$ = "abcdefghi"
little$ = "efg"
positionOfChar& = InStr(1, big$, little$)
Print positionOfChar& ' Output:  5
```

InStrB function

Returns the position of the byte beginning the first occurrence of one string within another string.

Syntax

InStrB ([*begin* **,**] *string1* **,** *string2* **)**

Elements

begin

Optional. A numeric expression with a positive integer value, *begin* specifies the character position in *string1* where InstrB should begin searching for *string2*. If you omit *begin*, it defaults to 1.

string1

The string to be searched.

string2

The string for which InStrB searches.

Return value

InStrB returns the byte position of the first occurrence of *string2* in *string1*. The following table shows how the function responds to various conditions.

Condition	*Return value*
string1 is " " (the empty string)	0
string2 is not found after *begin* in *string1*	0
begin is larger than the length of *string1*	0
string2 is " " (the empty string)	The value of *begin*. (If you omit *begin*, InStrB returns the value 1.)
string1 is NULL	NULL
string2 is NULL	NULL
begin is NULL	Error

Usage

If you want to work with characters, use the InStr function.

Note that the byte position returned by InStrB is independent of the platform-specific byte order.

Examples: InStrB function

```
' The value 9 (the position of the byte where the first
' occurrence of littleStr begins in bigStr) is assigned to
' the variable positionOfByte.

Dim bigStr As String, littleStr As String
Dim positionOfByte As Long
bigStr$ = "abcdefghi"
littleStr$ = "efg"
positionOfByte& = InStrB(1, bigStr$, littleStr$)
Print positionOfByte& ' Output:  9
```

InStrBP function

Returns the position of the byte (in the platform-native character set) beginning the first occurrence of one string within another string.

Syntax

InStrBP ([*begin* **,**] *string1* **,** *string2* **)**

Elements

begin

Optional. A numeric expression with a positive integer value, *begin* specifies the character position in *string1* where InstrBP should begin searching for *string2*. If you omit *begin*, it defaults to 1.

string1

The string to be searched.

string2

The string for which InStrBP searches.

Return value

InStrBP returns the byte position in the platform-specific character set of the first occurrence of *string2* in *string1*. The following table shows how the function responds to various conditions.

Condition	*Return value*
string1 is " " (the empty string)	0
string2 is not found after *begin* in *string1*	0
begin is larger than the length of *string1*	0
string2 is " "(the empty string)	The value of *begin*. (If you omit *begin*, InStrB returns the value 1.)
string1 is NULL	NULL
string2 is NULL	NULL
begin is NULL	Error

Usage

If you want to work with characters, use the InStr function.

Examples: InStrBP function

```
' The value 5 or other value depending on platform
' (the position of the byte where the first
' occurrence of littleStr begins in bigStr) is assigned to
' the variable positionOfByte.

Dim bigStr As String, littleStr As String
Dim positionOfByte As Long
bigStr$ = "abcdefghi"
littleStr$ = "efg"
positionOfByte& = InStrBP(1, bigStr$, littleStr$)
Print positionOfByte& ' Output:  9
```

Int function

Returns the nearest integer value that is less than or equal to a number.

Syntax

Int (*numExpr* **)**

Elements

numExpr

Any numeric expression.

Return value

The data type of *numExpr* determines the data type of the value returned by the Int function. The following table shows special cases.

numExpr	*Return value*
NULL	NULL
Variant containing a string interpretable as a number	Double

Usage

The value returned by the Int function is always less than or equal to its argument.

The Fix function and the Int function behave differently. Fix removes the fractional part of its argument, truncating toward 0.

Examples: Int function

```
Dim xF As Integer, yF As Integer
Dim xT As Integer, yT As Integer
xF% = Fix(-98.8)
yF% = Fix(98.2)
xT% = Int(-98.8)
yT% = Int(98.2)
Print xF%; yF%
' Output:
' -98  98
Print xT%; yT%
' Output:
' -99  98
```

Integer data type

Specifies a variable that contains a signed 2-byte integer.

Usage

An Integer value is a whole number in the range -32768 to 32767, inclusive.

Integer variables are initialized to 0.

The Integer suffix character for implicit type declaration is %.

LotusScript aligns Integer data on a 2-byte boundary. In user-defined data types, declaring variables in order from highest to lowest alignment boundaries makes the most efficient use of data storage space.

Examples: Integer data type

```
' The variable count is explicitly declared as type Integer.
' The variable nextOne is implicitly declared as type Integer
' by the % suffix character.
Dim count As Integer
count% = 1
nextOne% = count% + 1
Print count%; nextOne%          ' Output:   1  2
```

IsArray function

Tests the value of an expression to determine whether it is an array.

Syntax

IsArray (*expr* **)**

Elements

expr

Any expression.

Return value

IsArray returns TRUE (-1) if *expr* is an array; otherwise IsArray returns FALSE (0).

Examples: IsArray function

```
Dim arrayFixed(1 To 5)
Dim arrayDynam()
Print IsArray(arrayFixed)        ' Output:  True
Print IsArray(arrayDynam)        ' Output:  True

Dim v As Variant
Print IsArray(v)      ' Output:  False
v = arrayFixed
Print IsArray(v)      ' Output:  True
```

IsDate function

Tests the value of an expression to determine whether it is a date/time value.

Syntax

IsDate (*expr* **)**

Elements

expr

Any expression.

Return value

IsDate returns TRUE (-1) if *expr* is any of the following:

- A Variant value of DataType 7 (Date/Time)
- A Variant value of type String, where the string represents a valid date/time value
- A String value representing a valid date/time value

Otherwise IsDate returns FALSE (0).

Usage

A date/time value stored in a Variant is an 8-byte floating-point value. The integer part represents a serial day counted from Jan 1, 100 AD. Valid dates are represented by integers between -657434 (representing Jan 1, 100 AD) and 2958465 (representing Dec 31, 9999 AD). The fractional part represents the time as a fraction of a day, measured from time 00:00:00 (midnight on the previous day). In this representation of date/time values, day 1 is the date December 31, 1899.

Examples: IsDate function

```
Dim x As Variant, y As Variant, z As Variant
x = 100                          ' Numeric value
y = CDat(100)                    ' Numeric date value
z = "Nov 2, 1983"                ' String representing a date

Print IsDate(x)                  ' Output:  False
Print IsDate(y)                  ' Output:  True
Print IsDate(z)                  ' Output:  True
Print IsDate("100")              ' Output:  False
Print IsDate("Nov 2, 1983")      ' Output:  True
```

IsDefined function

Tests a string expression to determine whether it is the name of a product constant at run time.

Syntax

IsDefined (*stringExpr* **)**

Elements

stringExpr

Any string expression.

Return value

IsDefined returns TRUE (-1) if *stringExpr* is the name of a product constant at run time. Otherwise IsDefined returns FALSE (0).

Usage

The IsDefined function is used as a run-time parallel to the %If directive. It is commonly used to test the run-time value of a platform-identification constant that may be used to govern conditional compilation in a %If directive.

Note that IsDefined is not a LotusScript keyword.

IsElement function

Tests a string to determine whether it is a list tag for a given list.

Syntax

IsElement (*listName* **(** *stringExpr* **))**

Elements

listName

The name of a defined list.

expr
Any expression.

Return value

The IsElement function returns TRUE (-1) if *stringExpr* is the list tag for any element of *listName*. Otherwise IsElement returns FALSE (0).

Usage

If *listName* is not the name of a defined list, LotusScript generates an error.

If *expr* is a numeric expression, LotusScript first converts its value to a string.

If the character set is single byte, Option Compare determines whether list names are case sensitive. For example, if Option Compare Case is in effect, the names “ListA” and “Lista” are different; if Option Compare NoCase is in effect, these names are the same. If the character set is double byte, list names are always case and pitch sensitive.

Examples: IsElement function

```
' Use IsElement to determine whether
' the user correctly identifies a list tag.

' Declare a list to hold employee Ids.
Dim empList List As Double
Dim empName As String, Id As Double, found As Integer
' Create some list elements and assign them values.
empList#("Maria Jones") = 12345
empList#("Roman Minsky") = 23456
empList#("Joe Smith") = 34567
empList#("Sal Piccio") = 91234
' Ask the user to identify the list item to be removed.
empName$ = InputBox$("Which employee is leaving?")

' Check to see if empName$ corresponds to a list tag. If not, display
' a message and stop. Otherwise, validate the employee's Id.
' If everything checks out, remove the item from the list.
If IsElement(empList#(empName$)) = TRUE Then
   Id# = CDbl(InputBox$("What's " & empName$ & "'s Id?"))
   found% = FALSE          ' Initialize found to 0 (FALSE)
   ForAll empId In empList#
      If empId = Id# Then
         found% = TRUE     ' Set found to -1 (TRUE).
         If ListTag(empId) = empName$ Then
            Erase empList#(empName$)
            ' Verify the removal of the list element.
            If IsElement(empList#(empName$)) = FALSE Then
               MessageBox empName$ & _
                  " has been removed from the list."
            End If
         Else
```

```
            MessageBox "Employee name and Id do not match."
         End If
         ' No need to look farther for Id, so get out
         ' of the ForAll loop.
         Exit ForAll
      End If
   End ForAll
   If found% = FALSE Then
      MessageBox "Not a valid employee Id."
   End If
Else
   MessageBox "We have no such employee."
End If
```

IsEmpty function

Tests the value of an expression to determine whether it is EMPTY.

Syntax

IsEmpty (*expr* **)**

Elements

expr

Any expression.

Return value

The IsEmpty function returns TRUE (-1) if *expr* has the value EMPTY. This occurs only if *expr* is a Variant and has not been assigned a value.

Otherwise IsEmpty returns FALSE (0).

Examples: IsEmpty function

```
Dim dynaVar As Variant
Print IsEmpty(dynaVar)          ' Output:  True
dynaVar = PI
Print IsEmpty(dynaVar)          ' Output:  False
```

IsList function

Tests the value of an expression to determine whether it is a list.

Syntax

IsList (*expr* **)**

Elements

expr

Any expression.

Return value

The IsList function returns TRUE (-1) if *expr* is a list; otherwise IsList returns FALSE (0).

Examples: IsList function

```
Dim myList List
Print IsList(myList)     ' Output:  True

Dim v As Variant
Print IsList(v)          ' Output:  False
v = myList
Print IsList(v)          ' Output:  True
```

IsNull function

Tests the value of an expression to determine whether it is NULL.

Syntax

IsNull (*expr* **)**

Elements

expr

Any expression.

Return value

IsNull returns TRUE (-1) if *expr* is NULL; otherwise it returns FALSE (0).

Usage

The IsNull function checks whether a Variant contains NULL. For example:

```
If IsNull(LoVar) Then Print "LoVar is NULL" Else Print LoVar
```

Examples: IsNull function

```
Dim v As Variant
Print IsNull(v)                ' Output:  False
Print IsEmpty(v)               ' Output:  True
v = NULL
Print IsNull(v)                ' Output:  True
```

IsNumeric function

Tests the value of an expression to determine whether it is numeric, or can be converted to a numeric value.

Syntax

IsNumeric (*expr* **)**

Elements

expr

Any expression.

Return value

The IsNumeric function returns TRUE (-1) if the value of *expr* is a numeric value or can be converted to a numeric value. The following values are numeric:

- Integer
- Long
- Single
- Double
- Currency
- Date/Time
- EMPTY
- String (if interpretable as number)
- OLE error
- Boolean (TRUE, FALSE)

If *expr* is not a numeric value and cannot be converted to a numeric value, IsNumeric returns FALSE (0). The following values are not numeric:

- NULL
- Array
- List
- Object (OLE Automation object, product object, or user-defined object)
- String (if not interpretable as number)
- NOTHING

Usage

A common use of IsNumeric is to determine whether a Variant expression has a numeric value.

Examples: IsNumeric function

```
Dim v As Variant
Print IsNumeric(v)              ' Output:  True (v is EMPTY)
v = 12
Print IsNumeric(v)              ' Output:  True

' A string that is not interpretable as a number
v = "Twelve"
Print IsNumeric(v)              ' Output:  False

' A string that is interpretable as a number
v = "12"
Print IsNumeric(v)              ' Output:  True
```

IsObject function

Tests the value of an expression to determine whether it is a user-defined object, a product object, or an OLE Automation object.

Note IsObject is not supported under OS/2, under UNIX, or on the Macintosh.

Syntax

IsObject (*expr* **)**

Elements

expr

Any expression.

Return value

The IsObject function returns TRUE (-1) if the value of *expr* is an object (user-defined object, product object, or OLE Automation object) or NOTHING. Otherwise IsObject returns FALSE (0).

Examples: IsObject function

```
' Define two classes, Vegetable and Fruit.
Class Vegetable
   ' ... class definition
End Class
Class Fruit
   ' ... class definition
End Class

Dim tomato As Variant, turnip As Variant
Print IsObject(tomato)          ' Output:  False
Set turnip = New Vegetable
Print IsObject(turnip)          ' Output:  True
Set tomato = New Fruit
Print IsObject(tomato)          ' Output:  True
```

IsScalar function

Tests an expression to determine if it evaluates to a single value.

Syntax

IsScalar (*expr* **)**

Elements

expr

Any expression.

Return value

The IsScalar function returns TRUE (-1) if *expr* evaluates to one of the following:

- EMPTY
- Integer
- Long
- Single
- Double
- Currency
- Date/Time
- String
- OLE error
- Boolean (TRUE, FALSE)

Otherwise (if *expr* is an array, list, object, NOTHING, or NULL), IsScalar returns FALSE (0).

Examples: IsScalar function

```
Dim var As Variant
Print IsScalar(var)              ' Output:  True
var = 1
Print IsScalar(var)              ' Output:  True
var = "hello"
Print IsScalar(var)              ' Output:  True

Class SenClass
   ' ... class definition
End Class
Set var = New SenClass
Print IsScalar(var)       ' Output:  False
```

```
Dim senArray(1 To 5)
var = senArray
Print IsScalar(var)       ' Output:  False

Dim senList List
var = senList
Print IsScalar(var)       ' Output:  False
```

Kill statement

Deletes a file.

Syntax

Kill *fileName*

Elements

fileName

A string expression whose value is a file name; wildcards are not allowed. *fileName* can contain a drive indicator and path information.

Usage

Use Kill with care. If you delete a file with the Kill statement, you can't restore it with LotusScript statements or operating system commands. Make sure the file is closed before you attempt to delete it.

Kill deletes files, not directories. To remove directories, use the RmDir statement.

Examples: Kill statement

```
' Delete the file c:\test from the file system.
Kill "c:\test"
```

LBound function

Returns the lower bound for one dimension of an array.

Syntax

LBound (*arrayName* [**,** *dimension*] **)**

Elements

arrayName

The name of an array

dimension

Optional. An integer argument that specifies the array dimension; the default is 1.

Return value
The LBound function returns an Integer.

Usage
The default value for *dimension* is 1.

LotusScript sets the lower bound for each array dimension when you declare a fixed array or define the dimensions of a dynamic array with a ReDim statement.

The default lower bound for an array dimension is 0 or 1, depending on the Option Base setting.

Examples: LBound function
```
Dim minima(10 To 20)
Print LBound(minima)   ' Output:  10
```

LCase function

Returns the lowercase representation of a string.

Syntax
LCase[**$**] **(** *expr* **)**

Elements
expr

Any numeric or String expression for LCase; and any Variant or String expression for LCase$.

Return value
LCase returns a Variant of DataType 8 (a String), and LCase$ returns a String.

Usage
LCase ignores non-alphabetic characters.

LCase(NULL) returns NULL. LCase$(NULL) returns an error.

Examples: LCase function
```
Print LCase$("ABC")            ' Output:  "abc"
```

Left function

Extracts a specified number of the leftmost characters in a string.

Syntax
Left[**$**] **(** *expr* **,** *n* **)**

Elements

expr

Any numeric or String expression for Left; and any Variant or String expression for Left$. If *expr* is numeric, LotusScript converts it to a string before performing the extraction.

n

The number of characters to be returned.

Return value

Left returns a Variant of DataType 8 (a String), and Left$ returns a String.

If *n* is 0, the function returns the empty string (“”). If *n* is greater than the length (in characters) of *expr*, the function returns the entire string.

Left(NULL) returns NULL. Left$(NULL) is an error.

Examples: Left function

```
' Assign the leftmost 2 characters in "ABC".
Dim subString As String
subString$ = Left$("ABC", 2)
Print subString$                    ' Output:  "AB"
```

LeftB function

Lotus does not recommend using the LeftB function in LotusScript Release 3 because Release 3 uses Unicode, a character set encoding scheme that represents each character as two bytes. Because a two-byte character can be accompanied by leading or trailing zeroes, extracting characters by byte position no longer yields reliable results.

Use the Left function for left character set extractions instead.

LeftBP function

Extracts a specified number of the leftmost bytes in a string using the platform-specified character set.

Syntax

LeftBP[**$**] **(** *expr* **,** *n* **)**

Elements

expr

Any numeric or String expression for LeftBP; and any Variant or String expression for LeftBP$. If *expr* is numeric, LotusScript converts it to a string before performing the extraction.

n
The number of bytes to be returned using the platform-specified character set.

Return value
LeftBP returns a Variant of DataType 8 (a String), and LeftBP$ returns a String.

If *n* is 0, the function returns the empty string (""). If *n* is greater than the length (in bytes) of *expr*, the function returns the entire string.

LeftBP(NULL) returns NULL. LeftBP$(NULL) is an error.

If a double-byte character is divided, the character is not included.

Examples: LeftBP function
```
' The value "AB" or other value depending on platform
' is assigned to the variable subString.

Dim subString As String
subString = LeftBP$("ABC", 2)
Print subString$  ' Output:  "AB"
```

Len function

Returns the number of characters in a string, or the number of bytes used to hold a numeric value.

Syntax
Len ({ *stringExpr* | *variantExpr* | *numericExpr* | *typeName* } **)**

Elements
stringExpr
Any string expression.

variantExpr
Any Variant expression that includes a variable name.

numericExpr
The name of a variable, an element of an array, an element of a list, or a member variable of a user-defined data type or class. The data type of *numericExpr* is numeric.

typeName
An instance of a user-defined data type. It can be a simple variable of that data type, or an element of an array variable or a list variable of that data type.

Return value
For *stringExpr*, Len returns the number of characters in the string expression.

For *variantExpr,* Len returns the number of characters required to hold the value of *variantExpr* converted to a String.

For *numericExpr,* Len returns the number of bytes required to hold the contents of *numericExpr*.

For *typeName,* Len returns the number of bytes required to hold the contents of all the member variables, unless the user-defined data type includes Variant or variable-length String members. In that case, the length of the variable of the user-defined data type may not be the same as the sum of the lengths of its member variables.

Usage

In LotusScript Release 3, Len(NULL) generates an error. In previous releases of LotusScript, Len(NULL) returned NULL.

Len(*v*), where *v* is EMPTY, returns 0.

To determine the length of a string in bytes rather than in characters, use the LenB function. To determine the length of a string in bytes in the platform-native character set, use the LenBP function.

Examples: Len function

Example 1

```
' The length of a string, in characters
Dim theString As String
theString$ = "alphabet"
Print Len(theString$) ' Output:  8

' The number of bytes used to hold a Single variable
Dim singleVar As Single
Print Len(singleVar!) ' Output:  4
```

Example 2

```
' User-defined data type with variable-length String member
Type OrderInfo
   ordID As String * 6
   custName As String
End Type

' An instance of the user-defined data type
Dim ord As OrderInfo
ord.ordID$ = "OR1234"
ord.custName$ = "John R. Smith"

' Total length of the ord's members is 19.
Print Len(ord.ordID$) + Len(ord.custName)

' Length of ord is 16.
Print Len(ord)
```

LenB function

Returns the length of a string in bytes, or the number of bytes used to hold a variable.

Syntax

LenB ({ *stringExpr* | *variantExpr* | *numericExpr* | *typeName* } **)**

Elements

stringExpr

Any string expression.

variantExpr

Any Variant expression that includes a variable name.

numericExpr

The name of a variable, an element of an array, an element of a list, or a member variable of a user-defined data type or class. The data type of *numericExpr* is numeric.

typeName

An instance of a user-defined data type. It can be a simple variable of that data type, or an element of an array variable or a list variable of that data type.

Return value

For *stringExpr,* LenB returns the number of bytes in the string expression.

For *variantExpr,* LenB returns the number of bytes required to hold the value of *variantExpr* converted to a String.

For *numericExpr,* LenB returns the number of bytes required to hold the contents of *numericExpr*.

For *typeName,* LenB returns the number of bytes required to hold the contents of all the member variables, unless the user-defined data type includes Variant or variable-length String members. In that case, the length of the variable of the user-defined data type may not be the same as the sum of the lengths of its member variables.

Usage

In LotusScript Release 3, LenB(NULL) generates an error. In previous releases of LotusScript, LenB(NULL) returned NULL.

LenB(*v*), where *v* is EMPTY, returns 0.

To determine the length of a string in characters, use the Len function. To determine the length of a string in bytes in the platform-native character set, use the LenBP function.

Examples: LenB function

```
' The length of an 8-character string, in bytes
Dim theString As String
theString$ = "alphabet"
Print LenB(theString$)          ' Output:  16

' The number of bytes used to hold a Single variable
Dim singleVar As Single
Print LenB(singleVar!)          ' Output:  4
```

LenBP function

Returns the length of a string in bytes, or the number of bytes used to hold a variable, in the platform-native character set.

Syntax

LenBP ({ *stringExpr* | *variantExpr* | *numericExpr* | *typeName* } **)**

Elements

stringExpr
: Any string expression.

variantExpr
: Any Variant expression that includes a variable name.

numericExpr
: The name of a variable, an element of an array, an element of a list, or a member variable of a user-defined data type or class. The data type of *numericExpr* is numeric.

typeName
: An instance of a user-defined data type. It can be a simple variable of that data type, or an element of an array variable or a list variable of that data type.

Return value

For *stringExpr*, LenBP returns the number of bytes in the string expression.

For *variantExpr*, LenBP returns the number of bytes required to hold the value of *variantExpr* converted to a String.

For *numericExpr*, LenBP returns the number of bytes required to hold the contents of *numericExpr*.

For *typeName*, LenBP returns the number of bytes required to hold the contents of all the member variables, unless the user-defined data type includes Variant or variable-length String members. In that case, the length of the variable of the user-defined data type may not be the same as the sum of the lengths of its member variables.

Usage

LenBP(NULL) generates an error.

LenBP(*v*), where *v* is EMPTY, returns 0.

To determine the length of a string in characters, use the Len function. To determine the length of a string in bytes in the LotusScript internal character set, use the LenB function.

Let statement

Assigns a value to a variable.

Syntax

[**Let**] *variableID* **=** *expr*

Elements

Let

Optional. The Let statement is chiefly useful as a means of documenting an assignment statement. The absence of the Let keyword has no effect on the assignment.

variableID

A variable or variable element to which the value of *expr* is assigned. *variableID* can be of any data type that LotusScript recognizes, other than an object reference, an array, or a list. *variableID* can take any of these forms:

- *variableName*

  A non-array, non-list variable. The variable may not be an array or list variable, but it may be a Variant containing an array or list.

- *arrayName* **(***subscripts* **)**

  An array element. *arrayName* is an array variable or a Variant containing an array.

- *listName* **(***listTag***)**

  A list element. *listName* is a list variable or a Variant containing a list.

- *typeVar***.***memberVar*

  A member variable of a user-defined data type. *typeVar* is an instance of a user-defined data type. *typeVar* can be an element of an array or list. *memberVar* is a member variable of that user-defined data type. *memberVar* can be a scalar data type, a fixed array, or a Variant containing a scalar data type, an array, a list, or an object reference.

- *object*.*memberVar*
  object..*memberVar*
  Me.*memberVar*

  A member variable or property of a class. *object* is an expression whose value is an object reference. *memberVar* is a member variable or property of that class, or an element of an array member variable, or an element of a list member variable. Use **Me** only within a procedure defined within the class.

expr

Any expression except one whose value is an object reference. The *expr* must be of the same data type as *variableID*, or else must be convertible to the data type of *variableID*. The rules for data type conversion determine how (if at all) LotusScript converts the value of *expr* before assigning it to *variableID*.

Usage

LotusScript assigns the value of *expr* to the variable or variable element named by *variableID*.

Do not use the Let statement to assign an object reference to a variable. Use the Set statement to do that.

Examples: Let statement

```
' This example shows several cases of assignment.
' Wherever the keyword Let appears, it can be omitted without effect.

Dim a As Integer, b As Integer, c As Integer
Let a% = 2
Let b% = a%
Print b%                          ' Output:  2
Let c% = b% + 1
Print c%                          ' Output:  3

' Assign the value of b to an array element.
Dim devArray(3)
Let devArray(1) = b%
Print devArray(1)                 ' Output:  2

' Assign the value of c to a list element.
Dim devList List
Let devList("one") = c%
Print devList("one")              ' Output:  3

' For an instance of a user-defined data type,
' assign the value of c - a to a member variable.
Type DevType
   num As Integer
End Type
Dim inst As DevType
```

```
Let inst.num% = c% - a%
Print inst.num%                    ' Output:  1

' For an instance of a user-defined class,
' assign the value of a + b to a member variable.
Class DevClass
   Public num% As Integer
End Class
Set devObj = New DevClass
Let devObj.num% = a% + b%
Print devObj.num%                  ' Output:  4
```

Line Input # statement

Reads a line from a sequential file into a String or Variant variable.

Syntax

Line Input #*fileNumber* **,** *varName*

Elements

#*fileNumber*

The number assigned to the file when you opened it. A # sign must precede the file number.

varName

A String or Variant variable to hold the contents of one line of the file

Usage

Line Input # reads characters from a sequential file until it encounters a newline character. Line Input # does not read the newline character into the variable.

When reading a multi-line string from a sequential file, use the Input # statement, not the Line Input # statement.

Examples: Line Input # statement

```
' Display the contents of c:\config.sys a line at a time.

Dim text As String, fileNum As Integer
fileNum% = FreeFile()

Open "c:\config.sys" For Input As fileNum%
Do While Not EOF(fileNum%)
   Line Input #1, text$
   Print text$                     ' Prints one line of config.sys
Loop

Close fileNum%
```

ListTag function

Returns the name of the list element currently being processed by a ForAll statement.

Syntax

ListTag (*refVar* **)**

Elements

refVar

The reference variable in a ForAll list iteration loop.

Return value

ListTag returns a String that is the name of the list element currently referred to by *refVar*.

ListTag generates an error if *refVar* is not the reference variable specified in the ForAll statement.

If Option Compare NoCase is in effect and the character set is single byte, names are returned as all uppercase. Option Compare has no effect if the character set is double byte.

Usage

The ListTag function is valid only inside a ForAll block whose target is a list.

Examples: ListTag function

```
Dim loft List As Integer
loft%("first") = 0
loft%("second") = 1
loft%("third") = 2

' Print list tags for the elements of Loft,
' each on its own line.
ForAll i In Loft%
   Print ListTag(i)
End ForAll
' Output:
' first
' second
' third
```

LOC function

Returns the current position of the file pointer in a file.

Syntax

LOC (*fileNumber* **)**

Elements

fileNumber

The number assigned to the file when you opened it.

Return value

The following table presents the LOC return values for random, sequential, and binary files.

File type	*Return value*
Random	The number of the last record read from or written to the file. This is the file pointer position, minus 1.
Sequential	The byte position in the file, divided by 128 and truncated to an integer.
Binary	The position of the last byte read from or written to the file. This is the file pointer position, minus 1.

Examples: LOC function

```
Type PersonRecord
   empNumber As Integer
   empName As String *20
End Type

Dim rec1 As PersonRecord, rec2 As PersonRecord
Dim fileNum As Integer
Dim fileName As String
fileNum% = FreeFile()
fileName$ = "data.txt"

' Create a sample file.
Open fileName$ For Random As fileNum%

' Write at record 1.
rec1.empNumber% = 123
rec1.empName$ = "John Smith"
Put #fileNum%, 1, rec1
Print LOC(fileNum%)       ' Output:  1

' Write at record 2.
rec2.empNumber% = 456
rec2.empName$ = "Jane Doe"
Put #fileNum%, 2, rec2
Print LOC(fileNum%)       ' Output:  2

' Read from record 1.
Get #fileNum%, 1, rec2
Print LOC(fileNum%)       ' Output:  1

Close fileNum%
```

Lock and Unlock statements

Provide controlled access to files.

Syntax

Lock [**#**]*fileNumber* [**,** *recordNumber* | { [*start*] **To** *end* }]

Unlock [**#**]*fileNumber* [**,** *recordNumber* | { [*start*] **To** *end* }]

Elements

fileNumber

The number assigned to the file when you opened it.

recordNumber

In a random file, the number of the record that you want to lock or unlock. In a binary file, the byte that you want to lock or unlock. The first record in a random file is record number 1; the first byte in a binary file is byte number 1. LotusScript locks or unlocks only the specified record or byte.

In a sequential file, LotusScript locks or unlocks the whole file, regardless of value you specify for *recordNumber*.

start **To** *end*

In a random file, the range of record numbers you want to lock or unlock. In a binary file, the range of bytes that you want to lock or unlock. If you omit *start*, LotusScript locks records or bytes from the beginning of the file to the specified *end* position. In a sequential file, LotusScript locks or unlocks the whole file, regardless of the *start* and *end* values.

Usage

In Windows 3.1, you must run SHARE.EXE to enable the locking feature if you are using MS-DOS® version 3.1 or later. Earlier versions of MS-DOS do not support Lock and Unlock.

Always use Lock and Unlock statements in pairs whose elements — *fileNumber*, *recordNumber*, *start*, and *end* — match exactly. If you do not remove all locks, or if the elements do not match exactly, unpredictable results can occur.

Examples: Lock and unlock statements

```
Type PersonRecord
   empNumber As Integer
   empName As String * 20
End Type

Dim rec1 As PersonRecord, rec2 As PersonRecord
Dim fileNum As Integer, recNum As Integer
Dim fileName As String
recNum% = 1
```

```
fileNum% = FreeFile()
fileName$ = "data.txt"

' Create a record.
Open fileName$ For Random As fileNum%
rec1.empNumber% = 123
rec1.empName$ = "John Smith"
Put #fileNum, recNum%, rec1
Print rec1.empName$ ; rec1.empNumber%
' Output:
' John Smith           123

' Lock and update the record.
Lock #fileNum%, recNum%
Get #fileNum%, recNum%, rec2
Print rec2.empName$ ; rec2.empNumber%
' Output:
' John Smith           123
rec2.empName$ = "John Doe"
Put #fileNum%, recNum%, rec2
Print rec2.empName$ ; rec2.empNumber%
' Output:
' John Doe             123

' Release the lock.
Unlock #fileNum%, recNum%
Close fileNum%
```

LOF function

Returns the length of an open file in bytes.

Syntax

LOF (*fileNumber* **)**

Elements

fileNumber

The number assigned to the file when you opened it.

Return value

The LOF function returns a value of type Long.

Usage

LOF works only on an open file. To find the length of a file that isn't open, use the FileLen function.

Examples: LOF function

```
Dim izFile As Integer
Dim fileName As String, fileContents as String

izFile% = FreeFile()
fileName$ = "c:\autoexec.bat"
Open fileName$ For Input As izFile%

' Use LOF to find the file length, and Input$ to read
' the entire file into the string veriable izFile.
fileContents$ = Input$(LOF(izFile%), izFile%)Log
Print fileContents$                    ' Display the file contents.
```

Log function in LotusScript

Returns the natural (base *e*) logarithm of a number.

Syntax

Log (*numExpr* **)**

Elements

numExpr

Any numeric expression greater than zero.

Return value

The Log function returns a value of type Double.

Usage

The base for natural logarithms (*e*) is approximately 2.71828.

Examples: LOF function

```
Dim izFile As Integer
Dim fileName As String, fileContents as String

izFile% = FreeFile()
fileName$ = "c:\autoexec.bat"
Open fileName$ For Input As izFile%

' Use LOF to find the file length, and Input$ to read
' the entire file into the string veriable izFile.
fileContents$ = Input$(LOF(izFile%), izFile%)
Print fileContents$                    ' Display the file contents.
```

Log function

Returns the natural (base *e*) logarithm of a number.

Syntax
Log (*numExpr* **)**

Elements
numExpr
Any numeric expression greater than zero.

Return value
The Log function returns a value of type Double.

Usage
The base for natural logarithms (*e*) is approximately 2.71828.

Examples: Log function

Example 1
```
Dim natLog As Double
natLog# = Log(18)                          ' Assigns 2.89037175789617
```

Example 2
```
' Compute the base 10 logarithm of a number.
Function Log10 (inVal As Single) As Single
   Log10 = Log(inVal!) / Log(10)
End Function

Print Log10(10)                 ' Output:  1
Print Log10(100)                ' Output:  2
Print Log10(1 / 100)            ' Output:  -2
Print Log10(1)                  ' Output:  0
```

Long data type

Specifies a variable that contains a signed 4-byte integer.

Usage
The Long suffix character is &.

Long variables are initialized to 0.

A Long value is a whole number in the range -2,147,483,648 to 2,147,483,647 inclusive.

LotusScript aligns Long data on a 4-byte boundary. In user-defined types, declaring variables in order from highest to lowest alignment boundaries makes the most efficient use of data storage space.

Examples: Long data type

```
' Explicitly declare a Long variable.
Dim particles As Long

' Implicitly declare a Long variable.
bigInt& = 2094070921

particles = bigInt&
Print bigInt&; particles        ' Output:  2094070921     2094070921
```

LSet statement

Assigns a specified string to a string variable and left-aligns the string in the variable.

Syntax

LSet *stringVar* **=** *stringExpr*

Elements

stringVar

The name of a string variable. It may be a fixed-length String variable, a variable-length String variable, or a Variant variable.

stringExpr

The string to be assigned to the variable and left-aligned.

Usage

If the length of *stringVar* is greater than the length of *stringExpr*, LotusScript left-aligns *stringExpr* in *stringVar* and sets the remaining characters in *stringExpr* to spaces.

If the length of *stringVar* is less than the length of *stringExpr*, LotusScript copies only that many of the leftmost characters from *stringExpr* to *stringVar*.

If *stringVar* contains a numeric value, LotusScript converts it to a string to determine the length of the result.

If *stringVar* is a Variant, it can't contain NULL.

You can't use LSet to assign values from an instance of one user-defined data type to another.

Examples: LSet statement

```
Dim x As Variant
x = "qq"            ' Length of x is 2
LSet x = "abc"      ' Assigns leftmost 2 characters
Print x             ' Prints "ab"
LSet x = "c"        ' Assigns "c" and pads on the right with a space,
                    ' because length of x is 2
Print x & "high"    ' Prints "c high"
x = "c"             ' Ordinary assignment; new length of x is 1
Print x & "high"    ' Prints "chigh"
```

LTrim function

Removes leading spaces from a string and returns the result.

Syntax
LTrim (*stringExpr* **)**

Elements
stringExpr
Any string expression.

Return value
LTrim returns the trimmed version of *stringExpr* without modifying the contents of *stringExpr* itself. LTrim returns a Variant of DataType 8 (a String), and LTrim$ returns a String.

Examples: LTrim function
```
Dim trimLeft$ As String
trimLeft$ = LTrim$("   abc ")
Print trimLeft$
Print Len(trimLeft$)
' Output:
' abc
'  4
' The string "abc " is assigned to trimLeft.
' Note that the trailing space was not removed.
```

MessageBox function and statement

Displays a message in a message box and waits for user acknowledgment. The function form returns a value corresponding to the button the user presses.

Function Syntax
MessageBox (*message* [**,** [*buttons* + *icon* + *default* + *mode*] [**,** *boxTitle*]] **)**

Statement Syntax
MessageBox *message* [**,** [*buttons* + *icon* + *default* + *mode*] [**,** *boxTitle*]]

The MessageBox function and statement are identical, except that only the function has a return value.

MsgBox is acceptable in place of MessageBox.

Elements

message

The message to be displayed in the message box (a string). *message* can be up to 512 characters in length.

buttons

Defines the number and type of buttons to be displayed in the message box:

Constant name	*Value*	*Buttons displayed*
MB_OK	0	OK
MB_OKCANCEL	1	OK and Cancel
MB_ABORTRETRYIGNORE	2	Abort, Retry, and Ignore
MB_YESNOCANCEL	3	Yes, No, and Cancel
MB_YESNO	4	Yes and No
MB_RETRYCANCEL	5	Retry and Cancel

icons

Defines the icons to be displayed in the message box:

Constant name	*Value*	*Icon displayed*
MB_ICONSTOP	16	Stop sign
MB_ICONQUESTION	32	Question mark
MB_ICONEXCLAMATION	48	Exclamation point
MB_ICONINFORMATION	64	Information

Examples: MessageBox function and statement

Example 1

```
' Display the message "Do you want to continue?"
' in a message box labeled "Continue?" and containing
' Yes and No buttons. Assign the return value from
' the MessageBox function to the variable answer.
%Include "lsconst.lss"
Dim boxType As Long, answer As Integer
boxType& = MB_YESNO + MB_ICONQUESTION
answer% = MessageBox("Do you want to continue?", boxType&, _
  "Continue?")
```

Example 2

```
' Use the MessageBox statement to display a
' multiline message in a message box labeled "Demo"
' and containing an OK button.
%Include "lsconst.lss"
Dim twoLiner$ As String
twoLiner$ = |This message
is on two lines|
MessageBox twoLiner$, MB_OK, "Demo"
```

Mid function

Extracts a string from within another string, beginning with the character at a specified position.

Syntax

Mid[**$**] **(** *expr* **,** *start* [**,** *length*] **)**

Elements

expr

Any numeric or string expression. LotusScript converts a numeric to a string before performing the extraction.

start

The position of the first character to extract from the string, counting from 1 for the leftmost character. The value of *start* must be between 1 and 32000, inclusive.

length

The number of characters to extract from the string. The value of *length* must be between 0 and 32000, inclusive.

Return value

Mid returns a Variant of DataType 8 (a string), and Mid$ returns a String.

If there are fewer than *length* characters in the string beginning at the *start* position, or if you omit the *length* argument, the function returns a string consisting of the characters from *start* to the end of *expr*.

If *start* is greater than the length of *expr*, the function returns the empty string (“”).

Examples: Mid function

```
Dim subString As String
subString$ = Mid$("ABCDEF", 2, 3)
Print subString$        ' Output: BCD
```

Mid statement

Replaces part or all of one string with characters from another string.

Syntax

Mid[**$**] **(** *stringVar* **,** *start* [**,** *length*] **) =** *stringExpr*

Elements

stringVar

A String variable, or a Variant variable containing a string value. The *stringVar* cannot be a literal string.

start

The position of the first character in *stringVar* that you want to replace. This value must be between 1 and 32000, inclusive.

length

Optional. The number of characters you want to use from *stringExpr*. This value must fall between 1 and 64000, inclusive.

stringExpr

A string expression. Characters from *stringExpr* replace characters in *stringVar*.

Usage

Mid can alter the size of *stringVar* in bytes if you are working with multibyte characters. For example, if you are replacing a single-byte character with a double-byte character, the size of the string in bytes increases.

Otherwise, Mid does not alter the length of *stringVar*. That is, Mid does not append characters to *stringVar*. Mid uses as many characters of *stringExpr* as will fit in *stringVar* beginning at *start* and ending at *start* + *length* – 1.

To direct Mid to use all of *stringExpr*, either omit *length*, or specify a *length* greater than the length of the value in *stringExpr*.

If *start* is greater than the length of *stringVar*, LotusScript generates an error.

Examples: Mid statement

```
Dim string1 As String, string2 As String
string1$ = "ABCDEF"
string2$ = "12345"
' Replace the characters "BCD" in string1
' with the characters "123" in string2.
Mid$(string1$, 2, 3) = string2$
Print string1$        ' Output: A123EF
```

The three-character string “BCD”, beginning at the second character of string1, is replaced with the first three characters contained in string2, “123”.

MidB function

Lotus does not recommend using MidB in LotusScript Release 3 because Release 3 uses Unicode, a character set encoding scheme that represents each character as two bytes. Because a two-byte character can be accompanied by leading or trailing zeroes, extracting characters by byte position no longer yields reliable results.

Instead, use the Mid function for character set extractions.

MidB statement

Lotus does not recommend using MidB statements in LotusScript Release 3 because Release 3 uses Unicode, a character set encoding scheme that represents each character as two bytes. This means that a character can be accompanied by leading or trailing zeroes.

Instead, use the Mid statement for character set replacement.

MidBP function

Extracts a number of bytes (using the platform-specified character set) from within another string, beginning at a specified position.

Syntax

MidBP[**$**] **(** *expr* **,** *start* [**,** *length*] **)**

Elements

expr

Any numeric or String expression for MidBP; and any Variant or String expression for MidBP$. If *expr* is numeric, LotusScript converts it to a string before performing the extraction.

start

The position of the first byte in *expr* that you want to return. This value must be between 1 and 64000, inclusive.

length

Optional. The number of characters you want to use from *expr*. This value must fall between 1 and 64000, inclusive.

Return value
MidBP returns a Variant of DataType 8 (a String), and LeftBP$ returns a String.

If there are fewer than *length* bytes in the string beginning at the *start* position, or if you omit the *length* argument, the function returns a string consisting of the characters from *start* to to the end of *expr*.

If *start* is greater than the length in bytes of *expr*, the function returns an empty string.

If a double-byte character is divided, the character is not included.

Examples: MidBP function
```
' The value "BCD" or other value depending on platform
' is returned.

Print MidBP("ABCDE"; 2; 3)
```

Minute function

Returns the minute of the hour (an integer from 0 to 59) for a date/time argument.

Syntax
Minute (*dateExpr* **)**

Elements
dateExpr

Any of the following kinds of expression:

- A valid date/time string of type String or Variant. LotusScript interprets a 2-digit designation of a year in a date/time string as that year in the twentieth century. For example, 17 and 1917 are equivalent year designations.
- A numeric expression whose value is a Variant of DataType 7 (Date/Time)
- A number within the valid date range: the range -657434 (representing Jan 1, 100 AD) to 2958465 (Dec 31, 9999 AD), inclusive
- NULL

Return value
Minute returns an integer between 0 and 59.

The data type of the return value is a Variant of DataType 2 (Integer).

Minute(NULL) returns NULL.

Examples: Minute function

```
' Construct a message that displays the current time and
' the number of hours, minutes, and seconds remaining in the day.
Dim timeFrag As String, hoursFrag As String
Dim minutesFrag As String, secondsFrag As String
Dim crlf As String, message As String
timeFrag$ = Format(Time, "h:mm:ss AM/PM")
hoursFrag$ = Str(23 - Hour(Time))
minutesFrag$ = Str(59 - Minute(Time))
secondsFrag$ = Str(60 - Second(Time))
crlf$ = Chr(13) & Chr(10)              ' Carriage return/line feed
message$ = "Current time: " & timeFrag$ & ". " & crlf$ _
   & "Time remaining in the day: " _
   & hoursFrag$ & " hours, " _
   & minutesFrag$ & " minutes, and " _
   & secondsFrag$ & " seconds."
MessageBox(message$)
```

MkDir statement

Creates a directory.

Syntax

MkDir *path*

Elements

path

A string expression whose value is the name of the directory you want to create.

Usage

A drive letter in *path* is optional. If it is not included, the current path is used, including the current drive and directory.

Use the path syntax for the platform on which you are running LotusScript. The maximum allowable length of the *path* string varies with the platform.

LotusScript generates an error if the directory cannot be created.

Examples: MkDir statement

```
' Create directory TEST, in the root directory of drive C.
MkDir "c:\test"
```

Month function

Returns the month of the year (an integer from 1 to 12) for a date/time argument.

Syntax
Month (*dateExpr* **)**

Elements
dateExpr

Any of the following kinds of expression:

- A valid date/time string of String or Variant data type. LotusScript interprets a two-digit designation of a year in a date/time string as that year in the twentieth century. For example, 17 and 1917 are equivalent year designations.
- A numeric expression whose value is a Variant of DataType 7 (Date/Time)
- A number within the valid date range: the range -657434 (representing Jan 1, 100 AD) to 2958465 (Dec 31, 9999 AD), inclusive
- NULL

Return value
Month returns an integer between 1 and 12.

The data type of the return value is a Variant of DataType 2 (Integer).

Month(NULL) returns NULL.

Examples: Month function
```
Dim x As Long
Dim mm As Integer
x& = DateNumber(1994, 4, 1)
mm% = Month(x&)
Print mm%
' Output:
' 4
```

Name statement

Renames a file or directory.

Syntax
Name *oldName* **As** *newName*

Elements

oldName

A string expression whose value is the name of an existing file or directory, optionally including a path.

newName

A string expression whose value is the name to be given to the file or directory, optionally including a path. The *newName* cannot be another file or directory that already exists.

Usage

To move a file, specify complete paths in both *oldName* and *newName*. Use the same file name for both arguments if you don't want to rename it.

You can't move a file from one drive to another except under Windows NT and Windows 95.

You can't rename a file or directory to itself except under Windows NT and Windows 95.

You can rename a directory, but you can't move it.

You can't rename the current directory.

Examples: Name statement

```
' Rename the file WINDOWS\TEST1 to TEST2 and
' move it to the root directory of drive C.
Name "C:\WINDOWS\TEST1" As "C:\TEST2"
```

Now function

Returns the current system date and time as a date/time value.

Syntax

Now

Return value

Now returns the current system date and time as a Variant of DataType 7 (Date/Time).

Usage

A date/time value is an eight-byte floating-point value. The integer part represents a serial day counted from the date January 1, 100 AD. The fractional part represents the time as a fraction of a day, measured from midnight on the preceding day.

You can call the function as either Now or Now().

Examples: Now function

```
' Display the current date and time in the Long Date format
' (in Windows 3.1, determined by the system's LongDate
' International setting).

Print Format(Now(), "Long Date")
' Output:
' Tuesday, June 06, 1995
```

Oct function

Returns the octal representation of a number as a string.

Syntax

Oct[**$**] **(** *numExpr* **)**

Elements

numExpr

Any numeric expression. If *numExpr* evaluates to a number with a fractional part, LotusScript rounds it to the nearest integer before deriving its octal representation.

Return value

Oct returns a Variant of DataType 8 (String), and Oct$ returns a String.

Return values will only include the numerals 0 - 7, inclusive. The maximum length of the return value is 11 characters.

Usage

If the data type of *numExpr* is not Integer or Long, then LotusScript attempts to convert it to a Long. If it cannot be converted, a type mismatch error occurs.

Examples: Oct function

```
Print Oct$(17)                                  ' Prints "21"

' Converts Double argument to Long.
Print Oct$(17.0)                                ' Prints "21"

' Rounds Double argument, then converts to Long.
Print Oct$(17.3)                                ' Prints "21"

' Computes product 16.587, rounds to 17.0, then converts to Long.
Print Oct$(17.1 * .97)                          ' Prints "21"
```

On Error statement

Determines how an error will be handled in the current procedure.

Syntax

On Error [*errNumber*] { **GoTo** *label* | **Resume Next** | **GoTo 0** }

Elements

errNumber

Optional. An expression whose value is an Integer error number. If this is omitted, this statement refers to all errors in the current procedure. This value can be any error number that is defined in LotusScript at the time the On Error statement is encountered.

GoTo *label*

Specifies that when the error *errNumber* occurs, execution continues with an error handling routine that begins at *label*. The error is considered handled.

Resume Next

Specifies that when the error *errNumber* occurs, execution continues with the statement following the statement which caused the error. No error handling routine is executed. The values of the Err, Erl, and Error functions are not reset. (Note that a Resume statement does reset these values.) The error is considered handled.

GoTo 0

Specifies that when the error *errNumber* occurs, the error should not be handled in the current procedure. If *errNumber* is omitted, no errors are handled in the current procedure.

Usage

The On Error statement is an executable statement. It allows the procedure containing it to change the way LotusScript responds to particular errors. If no On Error statement is used, an error ordinarily causes execution to end. On Error allows a procedure to handle the error and continue execution appropriately.

How does On Error work?

An On Error statement is in effect from the time the statement runs until the procedure that contains it returns control to the calling program or procedure:

- If a procedure includes several On Error *errNumber* statements with the same error number, only the most recently executed one is in effect for that error number.
- The most recently executed On Error statement (with no *errNumber* element) is in effect for that error number if there is no On Error *errNumber* statement for that error number.

- If no On Error statement (without an *errNumber* element) has been executed, then the current procedure doesn't handle the error.

  In this case, LotusScript seeks an On Error statement for the error in the procedure's calling procedure, following the same rules for applying an On Error statement. If the caller doesn't handle the error, LotusScript looks in the caller's caller. If no applicable On Error statement is found by this process, execution ends, and the error message for the error is printed to the output window.

How does the error handling routine work?

An error handling routine begins with a labeled statement. The routine ends when LotusScript encounters a Resume, Exit Sub, Exit Property, or Exit Function statement. If an error occurs in the error handling routine, execution ends.

While the error handling routine is running, the Err, Erl, and Error functions describe the error being handled. A Resume statement will reset these values.

Where are error numbers and messages defined?

LotusScript specifies a standard set of errors, and corresponding error numbers (as constants), in the file lserr.lss. To define these errors and their numbers, include this file (using %Include) in a script that you compile or load before running any other script. Then these error numbers can be used in On Error statements to control error handling in the session.

Use the Error statement to define new error numbers and messages.

Examples: On Error statement

In this example, the On Error statement directs LotusScript to continue execution at the next statement after any error that occurs while the function Best is running.

The Call statement generates a division-by-zero error at the attempted division of y by z. Execution resumes at the next statement, the If statement. The current error number is the value of the constant ErrDivisionByZero, which was defined in the file lserr.lss previously included in the script by the %Include statement. Therefore the Print statement is executed. Then the Exit Function statement terminates execution within Best(), without executing further statements within the procedure; and control returns to the caller.

```
%Include "lserr.lss"
Function Best()
   Dim x As Integer, y As Integer, z As Integer

   ' After any error-generating statement, resume
   ' execution with the next statement.
   On Error Resume Next
   ' ...
   y% = 3
   z% = 0
   ' ...
```

```
   x% = y% / z%  ' Generates division-by-zero error.
   If Err = ErrDivisionByZero Then
      Print "Attempt to divide by 0. Returning to caller."
      Exit Function
   End If
   ' ...
End Function
Call Best()
```

On Event statement

Binds an event-handling sub or function to an event associated with a Lotus product object, or breaks an existing binding.

Note The Lotus product may provide an empty sub or function for each object event, in which case you do not need to use On Event statements. You can enter a script in the appropriate sub or function, and the script automatically executes when the event occurs. For details, see the product documentation.

Syntax

On Event *eventName* **From** *prodObject* { **Call** *handlerName* | **Remove** [*handlerName*] }

Elements

eventName

The name of an event specified in the product class definition.

prodObject

An expression whose value is a reference to a product object. (Events cannot be specified in user-defined class definitions.)

Call

Binds the *handlerName* sub or function to the specified *eventName* from the specified *prodObject*.

handlerName

The name of an event-handling sub or function for the specified *eventName* and *prodObject*. Whenever the specified event happens on the specified object, *handlerName* is called.

Remove

Detaches the *handlerName* sub or function from the object-event pair. If no *handlerName* is specified, this statement detaches all event-handling subs from the object-event pair.

Usage

An event-handling sub or function is defined like any other sub or function, with the restriction that its first parameter must be a reference to the product object that can raise the event. The remaining parameters are defined by the event in the product class, and are used in the handler call.

You can specify multiple event-handling subs or functions for the same event from the same object, using multiple On Event statements. The order of execution of event-handling subs or functions bound to the same event is undefined.

A function is necessary only if the event requires a return value from the handler.

Examples: On Event statement

This Lotus Forms example displays page 2 or 3 of the performanceRev form when the user clicks the buttonTurnPage command button, depending on the name of the current route stop. The first version uses an object event script provided by Lotus Forms. The second version uses On Event statements and event handler subs. For more information, see the Lotus Forms documentation.

```
' Version 1: Uses the command button Select event sub provided by
' Lotus Forms.

' Here are declarations for the entire form
Dim performanceRev As Form
Sub BUTTONbuttonTurnPage (b1 As Button)
   Set performanceRev = Bind Form ("")
   If performanceRev.RouteStopName = "Supervisor" Then
      performanceRev.GoToPage(2)
   Else
      performanceRev.GoToPage(3)
   End If
End Sub

' Version 2: The OpenForm event script uses On Event statements
' to call the appropriate event handler sub.
' The buttonTurnPage Select event script is empty.

' Here are declarations for the entire form.
Dim performanceRev As Form

' HandlerPage2 and HandlerPage3 are user-defined general subs with
' form-wide scope.
Sub HandlerPage2(B1 As Button)
   performanceRev.GoToPage(2)
End Sub
Sub HandlerPage3(B1 As Button)
   performanceRev.GoToPage(3)
End Sub

Sub FormOpenScript(F1 As Form)
   Set performanceRev = Bind Form("")
```

```
   If performanceRev.RouteStopName = "Supervisor" Then
      On Event Click From buttonTurnPage Call HandlerPage2
   Else
      On Event Click From buttonTurnPage Call HandlerPage3
   End If
End Sub

Sub BUTTONbuttonTurnPage (B1 As Button) ' This sub is not used.
End Sub
```

On...GoSub statement

Transfers control to one of a list of labels, processes statements until a Return statement is reached, and returns control to the statement immediately following the On...GoSub statement.

Syntax

On *numExpr* **GoSub** *label* [**,** *label*, ...]

Elements

numExpr

A numeric expression whose value determines which of the labels is the target of the transfer of control. The value of *numExpr* must not exceed 255.

label

A label that specifies the location of a series of statements to execute. The last statement in this series is a Return statement.

Usage

The On...GoSub statement, its labels, and the Return statement must all reside in the same procedure.

LotusScript transfers control to the first *label* if *numExpr* is 1, to the second *label* if *numExpr* is 2, and so on. Execution continues from the appropriate label until a Return statement executes. Then control returns to the statement immediately following the On...GoSub statement. If LotusScript encounters a statement (such as Exit or GoTo) that forces an early exit from the procedure before reaching a Return statement, the Return statement is not executed.

LotusScript rounds *numExpr* to the nearest integer before using it to determine the target label. If *numExpr* is 0, or is larger than the number of labels in the list, the On...GoSub statement is ignored and execution continues at the statement that immediately follows it.

LotusScript generates an error if *numExpr* evaluates to a number less than 0 or greater than 255.

Examples: On...GoSub statement

```
' The On...GoSub statement transfers control to Label3 and
' "Went to Label 3" is printed. Then control is returned to the
' statement following the On...GoSub statement, and
' "Successful return" is printed.
Sub Cleanup
   Dim x As Integer
   x% = 3
   On x% GoSub Label1, Label2, Label3
   Print "Successful return"   ' This prints
   Exit Sub
Label1:
   Print "Error"               ' This does not print
   Return
Label2:
   Print "Error"               ' This does not print
   Return
Label3:
   Print "Went to Label 3"     ' This prints
   Return
End Sub
```

On...GoTo statement

Transfers control to one of a list of labels.

Syntax

On *numExpr* **GoTo** *label* [**,** *label*]...

Elements

numExpr

A numeric expression whose value determines which of the labels is the target of the transfer of control. The value of *numExpr* must not exceed 255.

label

A label that specifies where control is to be transferred.

Usage

On...GoTo can't be used at the module level or to transfer control into or out of a procedure.

LotusScript transfers control to the first *label* if *numExpr* is 1, to the second *label* if *numExpr* is 2, and so on.

LotusScript rounds *numExpr* to the nearest integer before using it to determine the target label. If *numExpr* is 0, or is larger than the number of labels in the list, the On...GoTo statement is ignored and execution continues at the statement following it.

LotusScript generates an error if *numExpr* evaluates to a number greater than 255.

Examples: On...GoTo statement

This example illustrates On...GoTo and On Error.

The user enters a value. If the value is 1, 2, or 3, the On...GoTo statement transfers control to label1, label2, or label3. If the value is another number in the legal range for On...GoTo (the range is 0 - 255), control moves to the next statement. If the user enters a number that is out of range for On...GoTo, or that the CInt function cannot convert to an integer, an error occurs; and LotusScript transfers control to the OutOfRange label, in accordance with the On Error statement.

Depending on the user's entry, the OneTwoThree sub displays an appropriate message. If the entry is valid, an Exit Sub statement exits the Sub. If the entry is not valid, a GoTo statement transfers control to the EnterNum label, and the user is given another chance to make a valid entry.

```
Sub OneTwoThree
   Dim num As Integer
   On Error GoTo OutOfRange
EnterNum:
   num% = CInt(InputBox("Enter 1, 2, or 3"))
   On num% GoTo label1, label2, label3
  ' The user did not enter 1, 2, or 3, but a run-time error
  ' did not occur (the user entered a number in the range 0 - 255).
   MessageBox "You did not enter a correct value! Try again!"
   GoTo EnterNum
label1:
   MessageBox "You entered 1."
   Exit Sub
label2:
   MessageBox "You entered 2."
   Exit Sub
label3:
   MessageBox "You entered 3."
   Exit Sub
   ' An error condition has occurred.
OutOfRange:
   MessageBox "The value you entered is negative, " _
      & "greater than 255, or not a number. Try again!"
   GoTo EnterNum
End Sub
OneTwoThree ' Call the OneTwoThree sub.
```

Open statement

Opens a file, enabling access to it for reading or writing data.

Syntax

Open *fileName*

[**For** { **Random** | **Input** | **Output** | **Append** | **Binary** }]

[**Access** { **Read** | **Read Write** | **Write** }]

[{ **Shared | Lock Read** | **Lock Read Write** | **Lock Write** }]

As [**#**]*fileNumber*

[**Len =** *recLen*]

This statement must appear on one line, unless you use an underscore (_) for line continuation.

Elements

fileName

A string expression indicating the file to open. *fileName* may include a complete path. If you specify a *fileName* that does not exist, LotusScript generates an error if the mode is Input; for all other modes, LotusScript creates the file and opens it.

For *mode*

Optional. Specifies the file's mode; the default is Random.

- **Random**

  Default mode. Designates random access mode; that is, the file is accessible by record number. Use the Get and Put statements to read and write the file. If you omit the Access clause, LotusScript makes three attempts to open the file, using Read Write access, then Write access, and finally Read access. If all three attempts fail, an error is generated.

- **Input**

  Designates sequential input mode. Use the Input and Input # statements to read the file. If the mode conflicts with the Access type, LotusScript generates an error. For example, you can't open a file in Input mode with Write access.

  Output

  Designates sequential output mode. Use the Write # and Print # statements to write to the file. If the mode conflicts with the Access type, LotusScript generates an error. For example, you can't open a file in Output mode with Read access.

- **Append**

  Designates sequential output mode, beginning at the current end-of-file. If the mode conflicts with the Access type, LotusScript generates an error. For example, you can't open a file in Append mode with Read access. Unless you use the Seek statement to move to a file position other than the end of the file, the Print # and Write # statements append text to the end of the file.

- **Binary**

  Designates binary file mode. Use the Get and Put statements to read and write the file. If you omit the Access clause, LotusScript makes three attempts to open the file, using Read Write access, then Write access, and finally Read access. If all three attempts fail, an error is generated.

Access *operations*

Optional. Specifies what operations can be performed on the file. An error is generated if the access type conflicts with the file mode specified in the For clause.

- **Read**

  Default access type for Input mode. Only read operations are permitted.

- **Read Write**

  Default access type for Random mode. Both read and write operations are permitted.

- **Write**

  Default access type for Output, Append, and Binary modes. Only write operations are permitted.

Lock *type*

Optional. The default is Shared. Determines how the open file can be shared when accessed over a network by other processes, including processes owned by other users.

Under Windows 3.1, you must run SHARE.EXE to enable the locking feature if you are using MS-DOS version 3.1 or later. Lock is not supported for earlier versions of MS-DOS.

- **Shared**

  Default locking type. No file locking is performed. Any process on any machine on the network can read from or write to the file.

- **Lock Read**

  Prevents other processes from reading the file, although they can write to it. The lock is applied only if read access has not already been granted to another process.

- **Lock Read Write**

  Prevents other processes from reading and writing to the file. The lock is applied only if read or write access has not already been granted to another process. If a lock is already in place, it must be released before opening a file with Lock Read Write.

- **Lock Write**

  Prevents other processes from writing to the file, although they can read from it. The lock is applied only if write access has not already been granted to another process.

fileNumber

An integer expression with a value between 1 and 255, inclusive. This number is associated with the file when you open the file. Other file-manipulation commands use this number to refer to the file.

recLen

Optional. Designates the record length; use an integer expression with a value between 1 and 32767, inclusive.

For a Random file, *recLen* is the record length for the file (all records in a single file must have the same length). The default record length is 128 bytes.

For a sequential (Input, Output, or Append) file, *recLen* is the number of characters to be read from the file into an internal buffer, or assigned to an internal buffer before it is written to the file. This need not correspond to a record size, because the records in a sequential file can vary in size. A larger buffer uses more memory but provides faster file I/O. The default buffer size is 512 bytes.

For a Binary file, *recLen* is ignored.

Usage

If a file is already open in Binary, Random, or Input mode, you can open a copy of the file using a different file number, without closing the open file. If a file is already open in Append or Output mode, you must close it before opening it with a different file number.

LotusScript limits the number of open files to 255. Depending on your operating system environment and the Lotus product you are running, the actual number of files that you can open may be 15 or less. See your product documentation for details.

Examples: Open statement

```
' In this example, LotusScript reads the contents of a
' comma-delimited ASCII file (c:\123w\work\thenames.txt)
' into an array of RecType. RecType is a user-defined data type.
' c:\123w\work\thenames.txt consists of the following:
' "Maria Jones", 12345
' "Roman Minsky", 23456
' "Joe Smith", 34567
' "Sal Piccio", 91234

Type RecType
   empId As Double
   employee As String
End Type
Dim arrayOfRecs() As RecType
' A dynamic array that will get sized to
' the number of lines in c:\123w\work\thenames.txt
Dim txt As String
Dim fileNum As Integer
Dim counter As Integer
Dim countRec As Integer
' Get an unused file number so LotusScript can open a file.
fileNum% = FreeFile()
counter% = 0
Open "c:\123w\work\thenames.txt" For Input As fileNum%
Do While Not EOF(fileNum%)
   ' Read each line of the file.
   Line Input #fileNum%, txt$
   ' Increment the line count.
   counter% = counter% + 1
Loop
' Return the file pointer to the beginning of the file.
Seek fileNum%, 1
' The file has counter number of lines in it, so arrayOfRecs()
' is defined with that number of elements.
ReDim arrayOfRecs(1 To counter%)
' Read the contents of the file into arrayOfRecs.
For countRec% = 1 To counter%
   Input #fileNum%, arrayOfRecs(countRec%).employee$, _
      arrayOfRecs(countRec%).empId#
Next
Close fileNum%
Print arrayOfRecs(2).employee$ & " " arrayOfRecs(2).empId#
' Output:
' Roman Minsky 23456
```

Option Base statement

Sets the default lower bound for array subscripts to 0 or 1.

Syntax

Option Base *base*

Elements

base

The default lower bound (either 0 or 1) for all dimensions of all arrays in the module in which the Option Base statement occurs.

Usage

Option Base can be specified only once in a module, and only at the module level. If you use Option Base, it must precede all array declarations and all ReDim statements in the module.

The value set by Option Base applies to all arrays in the module that are either declared by Dim statements or redefined by ReDim statements.

If the module does not include an Option Base statement, the default lower bound for all dimensions of all arrays is 0. For example, a one-dimensional array of 10 elements would use subscripts 0 through 9.

Examples: Option Base statement

```
Option Base 1
' Create a one-dimensional array with 20 elements,
' which can be referred to as sample(1) to sample(20).
Dim sample(20) As Integer
```

Option Compare statement

Specifies the method of string comparison.

Syntax

Option Compare *option1* [**,** *option2*]

Elements

Option can be any of the following:

Binary

Comparison is bit-wise. If Binary is specified, no other option can be specified.

Case or **NoCase**

Comparison is case sensitive (default) or case insensitive. Only one of these options can be specified. The keyword Text is acceptable in place of NoCase.

Pitch or **NoPitch**

Comparison is pitch sensitive (default) or pitch insensitive. Only one of these options can be specified. These options apply to Asian (double byte) characters.

Usage

The Case, NoCase, Pitch, and NoPitch keywords specify string comparison using the character collation sequence determined by the Lotus product that you are using. The Binary keyword specifies string comparison in the platform's collation sequence: the effect is platform sort-order, case-sensitive, pitch-sensitive comparison.

Option Compare can be specified more than once per module, but the options cannot conflict. Option Compare can appear anywhere at module level. Option Compare applies to all string comparisons in the module. If you omit the Option Compare statement, the default method of string comparison is the same as Option Compare Case and Option Compare Pitch.

In certain functions such as InStr and StrCompare, the case and pitch sensitivity established by Option Compare or by default can be overridden by case-sensitivity and pitch-sensitivity arguments.

Examples: Option Compare statement

Example 1

In this example, the first call to function StrCompare uses the default (case-sensitive) setting without the optional argument that specifies a comparison method. In case-insensitive comparison, "A" equals "a", so StrCompare returns FALSE (0).

The second call to the function StrCompare specifies case-sensitive comparison in the country/language collation order, overriding the default established by Option Compare NoCase. In this comparison, "A" occurs earlier in the sort order than "a", so StrCompare returns TRUE (-1).

```
' The following results are for LotusScript in English,
' running on Windows 3.1.

Option Compare NoCase

' No method specified in StrCompare; use NoCase.
Print StrCompare("A", "a")   ' Output: False

' Use case-sensitive comparison
' (in country/language collation order).
Print StrCompare("A", "a", 0)   ' Output: True
```

Example 2

In this example, no Option Compare statement appears in the module, so the list tags "a" and "A" are different tags, because case-sensitive comparison in the country/language collation order is the default. Thus, the assignments to Loft("a") and Loft("A") refer to two different list elements. Within the ForAll statement, the ListTag function retrieves a list tag; and the Print statement prints it on a separate line.

```
Dim loft List As Integer
loft%("a") = 2
loft%("A") = 17
ForAll i In loft%
   Print ListTag(i)    ' Output: "a" and "A"
End ForAll
```

Example 3

In this example, the Option Compare NoCase statement specifies case-insensitive comparison in the country/language collation order as the default method for string comparison,. so the list tags "a" and "A" are the same tag. Thus, the assignments to loft("a") and loft("A") refer to the same list element. There is only one list tag for the ListTag function to retrieve and print.

```
Option Compare NoCase
Dim loft List As Integer
loft%("a") = 2
loft%("A") = 17
ForAll i In loft%
   Print ListTag(i)    ' Output: "A"
End ForAll
```

Example 4

In this example, the Option Compare Binary statement specifies bit-wise (platform sort-order, case-sensitive) comparison as the default method for string comparison, so the list tags "a" and "A" are different tags. Thus, the assignments to loft("a") and loft("A") refer to different list elements.

```
Option Compare Binary
Dim loft List As Integer
loft%("a") = 2
loft%("A") = 17
ForAll i In loft%
   Print ListTag(i)    ' Output: "a" and "A"
End ForAll
```

Option Declare statement

Disallows implicit declaration of variables.

Syntax
Option Declare

Explicit is acceptable in place of Declare.

Usage
Option Declare can be specified only once in a module, and only at the module level.

If the Option Declare statement appears in a module, then undeclared variables will generate syntax errors. When Option Declare is in effect, you must use the Dim statement to declare variables, except for arrays. You can still define an array implicitly using the ReDim statement.

Option Declare must be used before any variables are implicitly declared.

Examples: Option Declare statement
```
' Turn off implicit declaration of variables.
Option Declare
Dim y As Integer
y% = 10                ' No error
x = 20                 ' Compiler error (x has been not declared)
ReDim simAry(2, 2)     ' No error
```

Option Public statement

Specifies that module-level explicit declarations are Public by default.

Syntax
Option Public

Usage
Option Public can be specified only once in a module, and only at the module level. It must appear before any declarations in the module.

Option Public applies to module-level declarations for any variable, constant, procedure, user-defined data type, user-defined class, or external C function. It does not apply to label definitions, ForAll reference variables, or any implicitly declared variables.

The IDE automatically puts an Option Public statement in (Globals) (Options), so all (Globals) declarations are public by default. If you delete the Option Public statement, you must explicity specify the Public keyword to make (Globals) declarations public.

If a variable of a user-defined data type or an object reference variable is Public, the data type or the class to which it refers cannot be Private.

Use the Private keyword in a declaration to override Option Public for that declaration.

Examples: Option Public statement

```
' In this example, the Private keyword overrides Option Public in
' the declaration of the variables x, y, and z.
Option Public
Private x, y, z        ' x, y, and z are Private variables.
Dim i As Integer       ' i is Public.
```

Print statement

Prints data to the screen.

Syntax

Print [*exprList*]

Elements

exprList

A list of expressions separated by semicolons, spaces, or commas.

Usage

If *exprList* is omitted, Print prints a blank line.

Use the Spc and Tab functions to insert spaces and tabs between data items.

The Print statement adds a newline character to the end of *exprList* (to force a carriage return), unless *exprList* ends with a semicolon or a comma.

LotusScript inserts a '\n' character in any multiline string (for example, a string that you type in using vertical bars or braces). If you use Print to print the string, the \n is interpreted as a newline on all platforms.

The following table shows how the Print statement handles data items specified in *exprList*.

Data item	*Print statement behavior*
A variable	Prints the value of the variable.
A string	Prints the string.
A date/time value	Prints the date as a string in the operating system Short Date and Time format. If either the date part or the time part is missing from the value, only the supplied part is printed.
A Variant with the value EMPTY	Prints an empty string ("").
A Variant with the value Null	Prints the string "#NULL#".

The following table shows the effect of semicolons and commas in the Print statement.

Punctuation character	*Print statement behavior*
Semicolon or space in *exprList*	The next data item is printed with no spaces between it and the previous data item.
Semicolon at end of *exprList*	The next Print statement continues printing on the same line, with no spaces or carriage returns inserted.
Comma in *exprList*	The next data item is printed beginning at the next tab stop. (Tab stops are at every 14 characters.)
Comma at end of *exprList*	The next Print statement continues printing on the same line, beginning at the next tab stop. (Tab stops are at every 14 characters.)

Examples: Print statement

```
Dim a As Integer, b As Integer, c As Integer
a% = 5
b% = 10
c% = 15
Print a%, b%, c%        ' Prints 5  10  15
' LotusScript prints the values of a, b, and c, separating them
' with tabs and ending the line with a newline character.
```

Print # statement

Prints data to a sequential text file.

Syntax

Print #*fileNumber* **,** [*exprList*]

Elements

fileNumber

The file number assigned to the file when it was opened. Note that the pound sign (#), the file number, and the comma are all required.

exprList

Optional. A list of string and/or numeric expressions separated by semicolons, spaces, or commas. If you omit *exprList*, Print # prints a blank line. The maximum length of a string that can be printed is 32K characters.

Usage

Use Print # only on files opened in Output or Append mode. Unlike the Write # statement, the Print # statement does not separate the printed data items with formatting characters such as commas and quotation marks.

Use the Spc and Tab functions to insert spaces and tabs between data items.

If you set a width for the file using the Width statement, then the following occurs:

- A comma moves the next print position to the next tab stop. If this moves the print position past the defined width, the next data item is printed at the beginning of the next line.
- If the current print position is not at the beginning of a line and printing the next item would print beyond the defined width, the data item is printed at the beginning of the next line.
- If the item is larger than the defined width, it's printed anyway because Print # never truncates items. However, the line is terminated with a newline character to ensure that the next data item is printed on a new line.

The preceding statements about the effect of the Width statement apply for a width of 0, as well as any positive width.

The following table shows how the Print # statement handles data items specified in *exprList*.

Data item	*Print # statement behavior*
A variable	Prints the value of the variable.
A string	Prints the string.
A date/time value	Prints the date as a string in the operating system Short Date and Time format. If either the date part or the time part is missing from the value, only the supplied part is printed.
A Variant with the value EMPTY	Prints nothing to the file for the data item.
A Variant with the value Null	Prints the string "NULL" to the file.

The following table shows the effect of semicolons and commas in the Print # statement.

Punctuation character	*Print statement behavior*
Semicolon or space in *exprList*	The next data item is printed with no spaces between it and the previous data item.
Comma in *exprList*	The next data item is printed beginning at the next tab stop. (Tab stops are at every 14 characters.)

Examples: Print # statement

```
Dim nVar As Variant, eVar As Variant
nVar = NULL

Dim fileNum As Integer
fileNum% = FreeFile()
Open "printext.txt" For Output As fileNum%

' Print two lines to the file and close it.
' First line: two String values, with no separation between.
Print #fileNum%, "First line, " ; "with two String items"
' Second line: NULL value, EMPTY value, Integer variable value,
' and String value, all separated on the line by tabs.
Print #fileNum%, nVar, eVar, fileNum%, "at next tab"
Close fileNum%

' Open the file, print it, and close the file.
Dim text As String
Open "printext.txt" For Input As fileNum%
Do Until EOF(fileNum%)
   ' Read and print to console, one line at a time.
   Line Input #fileNum%, text$
   Print text$
Loop
Close fileNum%
' Output:
' First line, with two String items
' NULL                      1            at next tab
```

Property Get/Set statements

Define a property. A property is a named pair of Get and Set procedures that can be used as if they were a single variable.

Syntax

[**Static**] [**Public** | **Private**] **Property** { **Get | Set** } *propertyName* [**(** [*paramList*] **)**] [**As** *type*]

[*statements*]

End Property

Elements

Static

Optional. Specifies that the values of a Static property's variables are saved between calls to the property.

Public | **Private**

Optional. Public specifies that the property is visible outside the scope (module or class) where the property is defined, as long as this module is loaded. Private specifies that the property is visible only within the current scope.

A property in module scope is Private by default. A property in class scope is Public by default.

The Property Get and Property Set definitions for a property must use the same Public or Private setting.

Get | **Set**

Specifies which operation the procedure performs. A Property Get procedure retrieves the value of the property. A Property Set procedure assigns a value to the property.

propertyName

The name of the property. This name can have a data type suffix character appended to declare the data type of the value passed to and returned by the property.

paramList

Optional. A comma-separated list of declarations indicating the parameters to be passed to this property in Get and Set operations. The Get and Set operations must have the same number of arguments.

The syntax for each parameter declaration is:

[**ByVal**] *parameter* [**( )** | **List**] [**As** *type*]

ByVal means that *parameter* is passed by value: that is, the value assigned to *parameter* is a local copy of a value in memory, rather than a pointer to that value.

*parameter***()** is an array variable. *parameter* **List** identifies *parameter* as a list variable. Otherwise, *parameter* can be a variable of any of the other data types that LotusScript supports.

As *dataType* specifies the variable's data type. You can omit this clause and append a data type suffix character to *parameter* to declare the variable as one of the scalar data types. If you omit this clause and *parameter* has no data type suffix character appended (and isn't covered by an existing Def*type* statement), its data type is Variant.

Enclose the entire list of parameter declarations in parentheses.

type

Optional. The data type of values passed to and returned by the property.

type can be any of the scalar data types, a Variant, or a class name.

If **As** *Type* is not specified, the property name's data type suffix character determines the value's type. Do not specify both a *type* and a data type suffix character, as LotusScript treats that as an error.

If no *type* is specified and the property name has no data type suffix character appended, the property's value is either of data type Variant or of the data type specified by a Def*type* statement.

The *types* in the Property Get and Property Set definitions must be the same.

statements

Statements to retrieve or assign a property value.

Usage

The Public keyword cannot be used in a product object script or %Include file in a product object script, except to declare class members. You must put such Public declarations in (Globals).

A property usually consists of two procedures with the same name: a Property Get and a Property Set. However, you are not required to provide both.

A property member of a class cannot be declared Static. That is, a Property Get or Property Set statement within a class definition cannot begin with Static.

Using Property Get

A Property Get procedure is like a function. For example:

```
' These statements assign the value of saveInt to x
Dim saveInt As Integer
Property Get pInt As Integer
   pInt% = saveInt%
End Property
x = pInt%
```

Or:

```
' These statements assign the value of saveInt plus increment to x
Dim saveInt As Integer
Property Get pInt (increment As Integer) As Integer
   pInt% = saveInt% + increment%
End Property
x = pInt%(1%)
```

Using Property Set

A Property Set procedure is the reverse of a Property Get procedure. On entry into a Property Set procedure, an implicitly declared variable whose name and data type are the same as those of the Property Set procedure contains a value to be used inside the Property Set procedure. Inside the Property Set procedure, use the value of the variable instead of assigning a value to it.

Call a Property Set procedure by using its name on the left side of an assignment statement. The value on the right side of the statement is used by the Property Set procedure. For example:

```
' These statements assign the value of x to SaveInt
Dim SaveInt As Integer
Property Set pInt As Integer
   saveInt% = pInt%
End Property
pInt% = x
```

Or:

```
' These statements assign the value of x + increment to SaveInt
Dim SaveInt As Integer
Property Set pInt (increment As Integer) As Integer
   saveInt% = pInt% + increment%
End Property
pInt%(1%) = x
```

Referencing a property that returns an array, list, or collection

If a Get operation returns an array, list, or collection, a reference to the property can contain subscripts according to the following rules:

- If the property has parameters, the first parenthesized list following the reference must be the argument list. A second parenthesized list is treated as a subscript list. For example, p1(1,2)(3) is a reference to a property p1 that has two parameters and returns a container.
- If the property has no parameters and the return type is a variant or collection object, a single parenthesized list following the reference is treated as a subscript list. For example, p1(1) is a reference to a property p1 that either contains one parameter or contains no parameters, but is a container.

- If the property has no parameters and the return type is not a variant or collection object, any parenthesized list following the reference is an error, except that a single empty list is allowed. For example, p1() is a reference to a property p1 that contains no parameters and may or may not be a container; if p1 is a container, the reference is to the entire container.

In a Set operation, the property reference cannot be subscripted. A parenthesized list following the reference must be the argument list. For example, p1(1) is a reference to a property p1 with one parameter; p1(1,2)(3) or p1()(3) is illegal in a Set operation.

Passing a property to a function

A LotusScript property (a property defined by Property Get or Property Set) can be passed to a function by value only, not by reference.

Examples: Property Get/Set statements

```
' This example illustrates basic operations with a property.
' The counter is a property; it receives a starting value.
' Each time the property is used, it returns a value that is
' 1 greater than the previous value, until a new starting value
' is set. In this example, counter is set to 100. Then the property
' is used to print 101 and again to print 102.

' A variable to store values between uses of the property
Dim count As Integer

Property Get counter As Integer
   count% = count% + 1    ' Add 1 to the previous value.
   counter% = count%      ' Return the value.
End Property

Property Set counter As Integer
   count% = counter%      ' Assign the value to count.
End Property
counter% = 100

' Each time the property is used, it increments count
' by 1 and returns count's value.
Print counter%            ' Prints 101

Print counter%            ' Prints 102
```

Put statement

Writes data from a variable to a binary file or a random file.

Syntax

Put [**#**] *fileNumber* **,** [*recordNumber*] **,** *variableName*

Elements

fileNumber

The file number assigned to the file when it was opened with the Open statement. Note that the pound sign (#), *fileNumber,* and *variableName* are all required.

recordNumber

Optional. The file position (the byte position in a binary file, or the record number in a random file) where data is written. If you omit the *recordNumber,* data is written starting at the current file position.

variableName

The variable holding the data to be written. *variableName* cannot be an array; however, a fixed-length array defined within a data type is allowed (this array could even contain other arrays as elements).

Usage

The first byte or record in a file is always file position 1. After each write operation, the file position is advanced:

- For a binary file, by the size of the variable
- For a random file, by the size of a record

If *variableName* is shorter than the length of a record in the file, Put does not overwrite or delete any data that may already be stored in the remainder of that record.

The following table shows how the Put statement behaves for different data types.

variableName data type	*Put statement's behavior*
Variant	The Put statement writes the DataType as the first two bytes before the value itself. If the DataType is EMPTY or NULL, the Put statement writes no more data. If the DataType is numeric, the Put statement writes the number of bytes of data appropriate for that DataType: Integer: 2 bytes Long: 4 bytes Single: 4 bytes Double: 8 bytes Currency: 8 bytes Date/time: 8 bytes
Fixed-length String	The Put statement writes the specified number of characters. For example, if a variable is declared as String * 10, then exactly 10 characters are written.
Variable-length String	The Put statement behaves differently, depending on the type of file you're using. *Random files*: The first two bytes written indicate the length of the string. Then the Put statement writes the number of characters specified by that length. If *variableName* is not initialized, the Put statement writes a string of length 0. If *variableName* is longer than a record, LotusScript generates the "Bad record length" error. If *variableName* is shorter than a record, the remainder of the record is not cleared. *Binary files*: The number of bytes written to the file is equal to the length of the string currently stored in *variableName*. If *variableName* is not initialized, no data is written to the file. Note that in binary files, data is written without regard to record length.
User-defined data type	The Put statement writes the sum of the bytes required to write all members of the used-defined data type, which cannot contain a dynamic array, a list, or an object.

When Put writes out String data, the characters are always written in the Unicode character set.

Examples: Put statement

```
Type PersonRecord
   empNumber As Integer
   empName As String * 20
End Type

Dim fileNum As Integer
Dim fileName As String
Dim rec As PersonRecord
fileNum% = FreeFile()
fileName$ = "data.txt"

' First, open a random file with a record length equal to
' the size of the records to be stored.
Open fileName$ For Random As fileNum% Len = Len(rec)

rec.empNumber% = 123
rec.empName$ = "John Smith"
Put #fileNum%, 1, rec       ' Write this record at position 1.
rec.empNumber% = 456
rec.empName$ = "Jane Doe"
Put #fileNum%, 2, rec       ' Write this record at position 2.
rec.empNumber% = 789
rec.empName$ = "Jack Jones"
Put #fileNum%, , rec        ' Write at current position (3).

Seek fileNum%, 1            ' Rewind file to beginning.
Do While Not EOF(fileNum%)
   ' Get a record, print it out.
   ' Get advances the file position to the next record automatically.
   Get #fileNum%, , rec
   Print rec.empNumber%, rec.empName$
Loop
' Output:
' 123         John Smith
' 456         Jane Doe
' 789         Jack Jones
Close fileNum%              ' Close the file.
```

Randomize statement

Seeds (initializes) the random number generator.

Syntax

Randomize [*numExpr*]

Elements

numExpr

Any numeric expression. If you omit *numExpr*, Randomize uses the return value from Timer.

Usage

Use Randomize to seed the random number generator before calling Rnd to generate a number.

If you use Randomize with *numExpr* and then repeatedly call Rnd with no arguments, LotusScript returns the same sequence of random numbers every time you run the script. To generate a different sequence of random numbers each time you run the script, do one of the following:

- Use a variable *numExpr* to make sure that Randomize receives a different seed value every time the script is executed.
- Use Randomize with no *numExpr*. This seeds the random number generator with the return value from Timer.

The particular sequence of random numbers generated from a given seed depends on the platform where you are running LotusScript.

Examples: Randomize statement

Example 1

```
Randomize 17    ' Use 17 to seed the random number generator.
Print Rnd(); Rnd(); Rnd(); Rnd(); Rnd()
' Output:
' .9698573  .8850777  .8703259  .1019439  .7683496
' If you rerun this script (on the same platform), LotusScript
' generates the same sequence of random numbers,
' because the same seed is used.
```

Example 2

```
Randomize               ' Don't provide any seed.
Print Rnd(); Rnd(); Rnd(); Rnd(); Rnd()
                        ' Prints a series of random numbers.
' If you rerun this script, LotusScript produces a different sequence
' of random numbers, because Randomize is called with no argument.
```

ReDim statement

Declares a dynamic array and allocates storage for it, or resizes an existing dynamic array.

Syntax

ReDim [**Preserve**] *arrayName* **(** *bounds* **)** [**As** *type*]

[**,** *arrayName* **(** *bounds* **)** [**As** *type*]] ...

Elements

Preserve

Optional. If you've already declared *arrayName*, LotusScript preserves the values currently assigned to it. If you omit Preserve, LotusScript initializes all elements of the array, depending on the data type of the array variable.

Data type of array variable	*Initial value of array element*
Integer, Long, Single, Double, or Currency	0
Fixed-length String	A string of the specified length, filled with the Null character (Chr(0))
Variable-length String	The empty string ("")
Variant	EMPTY
Class	NOTHING
User-defined data type	The initial value of each element's own data type

arrayName

The name of an array to be declared or resized. The *arrayName* must designate an array; it cannot be a Variant variable containing an array.

bounds

A comma-separated list of dimension bounds for *arrayName*. Each set of dimension bounds has the following form:

[*lowerBound* **To**] *upperBound*

The *lowerBound* is the minimum subscript allowed for the dimension, and *upperBound* is the maximum. If you don't specify a *lowerBound*, the lower bound for the array dimension defaults to 0, unless the default lower bound has been changed to 1 using the Option Base statement.

Array bounds must fall in the range -32768 to 32767, inclusive.

type

Optional. A valid LotusScript data type, user-defined type, or class that specifies the data type of *arrayName*.

You cannot change the data type of an existing array. If *arrayName* was declared and *type* is specified in the current ReDim statement, *type* must match the original data type of *arrayName*.

Usage

A ReDim statement allocates storage for a dynamic array. You can resize the array with additional ReDim statements as often as you want. Each time you resize the array, LotusScript reallocates the storage for it.

Unlike a Dim statement, ReDim cannot specify an array as Private, Public, or Static. To specify a dynamic array with one of these characteristics, declare it first in a Dim statement. If you declare a dynamic array with a Dim statement, LotusScript doesn't allocate storage for the array elements. You can't actually use the array in your script until you allocate storage with ReDim.

Arrays can have up to 8 dimensions. The first ReDim statement for an array sets the number of dimensions for the array. Subsequent ReDim statements for the array can change the upper and lower bounds for each dimension, but not the number of dimensions.

If Preserve is specified, you can change only the upper bound of the last array dimension. Attempting to change any other bound results in an error.

Do not use ReDim on a fixed array (an array already declared and allocated by a Dim statement).

If you're using ForAll on a container variable that is an array of arrays, do not ReDim the reference variable (this generates the "Illegal ReDim" error).

Examples: ReDim statement

Example 1

```
' The array x has not been previously declared,
' so ReDim automatically assigns it the data type Variant.
ReDim x(5)
Print DataType(x(1))                  ' Prints 0.

' The Dim statement declares array y with the data type String.
Dim y() As String

' The ReDim statement can't change the data type of an existing array.
' If you specify a data type for array y in the ReDim statement,
' it must be String.

ReDim y(5) As String
Print DataType(y$(1))                 ' Prints 8.
```

Example 2

```
Option Base 1
' Declare a two-dimensional dynamic array, of Variant type.
ReDim markMar(2, 2)

' Assign a value to each element.
markMar(1, 1) = 1
markMar(2, 1) = 2
markMar(1, 2) = 3
markMar(2, 2) = 4

' Change the upper bound of the last dimension of markMar from 2 to 3,
' preserving the values already stored in existing elements
' of markMar.
ReDim Preserve markMar(2,3)

' Assign values to the additional elements of markMar.
markMar(1, 3) = 5
markMar(2, 3) = 6
```

Rem statement

Indicates a one-line comment in a script.

Syntax

Rem *text*

Elements

text

A one-line comment that LotusScript ignores.

Usage

The Rem statement indicates a comment or "remark" in the script.

The Rem statement need not be the first statement on a line, but it is the last: the LotusScript compiler ignores all text from the Rem keyword to the end of the current line. A line continuation character (an underscore) does not continue a Rem statement.

The apostophe (') has the same effect as the Rem keyword and can appear anywhere on a line without needing a colon (:) to separate the statements. As with Rem, LotusScript ignores everything after the apostrophe.

Examples: Rem statement

Example 1

```
Rem  This is a comment in the script.
' This is also a comment in the script.
```

Example 2

```
x = 5 : Rem The colon is required to separate statements.
x = 5   ' No colon is required before a single quote.
```

%Rem directive

Indicates one or more comment lines in a script.

Syntax

%Rem

text

%End Rem

Elements

text

One or more lines of text that LotusScript ignores.

Usage

The compiler ignores all text between %Rem and %End Rem, including text on the same line.

%Rem and %End Rem must each be the first text on a line (they may be preceded on the line by spaces or tabs). Each must be followed by one or more spaces, tabs, or newline characters before any more text appears.

%Rem...%End Rem blocks cannot be nested.

Note For compatibility with older versions of the language, LotusScript Release 3 accepts the directive %EndRem (with no space) in place of %End Rem.

Examples: %Rem directive

Example 1

```
' The compiler ignores the lines of text between %Rem and %End Rem,
' and the text on the line beginning %Rem.
' It also ignores the line containing the Rem statement.
%Rem   Note that all text on this line is ignored by the compiler.
   What follows is ignored by the compiler. It can contain comments or
non-working statements.
   Check(, 'This, for example, would have been a syntax error.
%End Rem   This text is ignored as well.
Rem Normal parsing and compilation continues from here.
```

Example 2

```
' %Rem blocks cannot be nested, so the second %Rem directive is
' illegal in the following.
%Rem
Comment line 1
Comment line 2
...
%Rem            ' Error
Comment line
...
%End Rem
%End Rem
```

Reset statement

Closes all open files, copying the data from each file to disk.

Syntax

Reset

Usage

Before closing the open files, Reset writes all internally buffered data to the files.

Examples: Reset statement

```
' All open files are closed and the contents of the operating
' system buffer are written to disk.
Reset
```

Resume statement

Directs LotusScript to resume script execution at a particular statement in a script, after an error has occurred.

Syntax

Resume [**0** | **Next** | *label*]

Elements

0

Resumes execution at the statement that caused the current error.

Next
Resumes execution at the statement following the statement that caused the current error.

label
Resumes execution at the specified label.

Usage
Use the Resume statement only in error-handling routines; once LotusScript executes the Resume statement, the error is considered handled.

Resume continues execution within the procedure where it resides. If the error occurred in a procedure called by the current procedure, and the called procedure didn't handle the error, then Resume assumes that the statement calling that procedure caused the error:

- Resume [0] directs LotusScript to execute the procedure-calling statement that produced the error again.

  Note that this may result in an infinite loop, where in every iteration, the procedure generates the error and is then called again.
- Resume Next directs LotusScript to resume execution at the statement following the procedure call.

The Resume statement resets the values of the Err, Erl, and Error functions.

Examples: Resume statement
```
Sub ResumeSub()
   On Error GoTo ErrHandler
   ' ...
   Error 1                    ' Intentionally raise an error.
   Error 10
   Error 100
   ' ...
   Exit Sub

ErrHandler:                   ' Error-handling routine
   Print "Error " & Err & " at line number" &Erl
   Resume Next                ' Resume the procedure.
End Sub
' The error-handling routine prints information about the current
' error. Then LotusScript resumes execution of the script at the
' statement following the statement that caused the current error.
```

Return statement

Transfers control to the statement following a GoSub or On...GoSub statement.

Syntax

Return

Usage

The GoSub and On...GoSub statements transfer control to a labeled statement within a procedure. Execution continues from this statement until a Return statement is encountered. LotusScript then transfers control to the first statement following the GoSub or On...GoSub statement. While executing the procedure, LotusScript can encounter a statement, such as Exit or GoTo, that forces an early exit from the procedure; in this case, the Return is not executed.

The GoSub or On...GoSub statement, its labels, and the Return statement must reside in the same procedure.

Examples: Return statement

```
' In response to user input, LotusScript transfers control to
' one of three labels, constructs an appropriate message, and
' continues execution at the statement following the GoSub.
Sub GetName
   Dim yourName As String, Message As String
   yourName$ = InputBox$("What is your name?")
   If yourName$ = "" Then           ' The user enters nothing.
      GoSub EmptyString
   ' A case-insensitive comparison
   ElseIf LCase(yourName$) = "john doe" Then
      GoSub JohnDoe
   Else
      Message$ = "Thanks, " & yourName$ _
         & ", for letting us know who you are."
   End If
   ' The Return statements return control to the next line.
   MessageBox Message$
   Exit Sub

EmptyString:
   yourName$ = "John Doe"
   Message$ = "Okay! As far as we're concerned, " _
      & "your name is " & yourName$ & ", and you're on the run!"
   Return

JohnDoe:
   Message$ = "We're on your trail, " & yourName$ _
      & ". We know you are wanted dead or alive!"
   Return
End Sub
GetName                             ' Call the GetName sub.
```

Right function

Extracts a specified number of the rightmost characters in a string.

Syntax

Right[**$**] **(** *expr* **,** *n* **)**

Elements

expr

Any numeric or String expression for Right; and any Variant or String expression for Right$. If the expression is numeric, it is first converted to a string.

n

The number of characters to be returned.

Return value

Right returns a Variant of DataType 8 (String), and Right$ returns a String.

If *n* is 0, Right returns the empty string (“”); if *n* is greater than the number of characters in *expr*, Right returns the entire string.

Right(NULL,1) returns NULL. Right$(NULL,1) returns an error.

Usage

LotusScript Release 3 represents characters with two bytes instead of one, so Lotus no longer recommends using the RightB function to work with bytes.

Examples: Right function

```
Dim subString As String
subString$ = Right$("ABCDEF", 3)
Print subString$                ' Prints "DEF"
```

RightB function

LotusScript Release 3 uses Unicode, a character set encoding scheme that represents each character as two bytes. This means that a character can be accompanied by leading or trailing zeroes, so Lotus no longer recommends using RightB to work with bytes.

Instead, use the Right function for right character set extractions.

RightBP function

Extracts a specified number of the rightmost bytes in a string using the platform-specified character set.

Syntax

RightBP[**$**] **(** *expr* **,** *n* **)**

Elements

expr

Any numeric or String expression for RightBP; and any Variant or String expression for RightBP$. If *expr* is numeric, LotusScript converts it to a string before performing the extraction.

n

The number of bytes to be returned using the platform-specified character set.

Return value

RightBP returns a Variant of DataType 8 (a String), and RightBP$ returns a String.

If *n* is 0, the function returns the empty string (“”). If *n* is greater than the length (in bytes) of *expr*, the function returns the entire string.

RightBP(NULL) returns NULL. RightBP$(NULL) is an error.

If a double-byte character is divided, the character is not included.

Examples: RightBP function

```
' The value "BC" or other value depending on platform
' is assigned to the variable subString.

Dim subString As String
subString = RightBP$("ABC", 2)
Print subString$  ' Output:  "BC"
```

RmDir statement

Removes a directory from the file system.

Syntax

RmDir *path*

Elements

path

A String expression specifying the path of the directory you want to remove.

Usage

The maximum length of *path* depends on the platform you are using.

If the directory named by *path* is not empty, RmDir generates an error.

Examples: RmDir statement

```
' Remove directory c:\test from the file system.
RmDir "c:\test"
```

Rnd function

Generates a random number greater than 0 and less than 1.

Syntax

Rnd [**(** *numExpr* **)**]

Elements

numExpr

Any numeric expression.

Return value

The return value is a number of data type Single. The following table shows how Rnd behaves, depending on the sign of *numExpr*.

Sign of numExpr	*Rnd behavior*
Positive	Returns the next random number in the sequence of random numbers generated from the value that most recently seeded the random number generator.
Zero (0)	Returns the random number most recently generated.
Negative	The random number generator is seeded again with *numExpr*. Rnd returns the first number in the sequence generated from that seed value.

Usage

Use Randomize to seed the random number generator before calling Rnd to generate the number.

If you use Randomize with an argument and then repeatedly call Rnd (with no arguments), LotusScript returns the same sequence of random numbers every time you execute the script. The particular sequence of random numbers generated from a given seed depends on the platform where you are running LotusScript.

If you use Randomize without an argument, LotusScript generates a different sequence of numbers each time you execute the script.

You can call the function with no arguments as either Rnd or Rnd( ).

Examples: Rnd function

```
Randomize -1
Print Rnd(); Rnd(); Rnd(); Rnd(); Rnd()
' Output:
' 7.548905E-02  .5189801  .7423341  .976239  .3883555
Randomize -1
Print Rnd(0)
' Output:
' .3142746
Print Rnd(); Rnd(); Rnd(); Rnd(); Rnd()
' Output:
' 7.548905E-02  .5189801  .7423341  .976239  .3883555
Print Rnd(-1)
' Output:
' .3142746
Print Rnd(-2); Rnd(0)
' Output:
' .6285492  .6285492
```

Round function

Rounds a number to a specified number of decimal places.

Syntax

Round (*numExpr* **,** *places* **)**

Elements

numExpr

Any numeric expression. The number to be rounded.

places

Any numeric expression representing the desired number of decimal places. If *places* is not an integer, it is converted to one.

Return value

Round returns a Double.

If the first non-significant digit is 5, and all subsequent digits are 0, the last significant digit is rounded to the nearest even digit. See the example that follows.

If *places* is negative, the number is rounded to *places* digits to the left of the decimal point. See the example that follows.

Examples: Round function

```
' Round to one decimal place.
Print Round(4.23, 1)                    ' Prints 4.2
Print Round(4.35, 1)                    ' Prints 4.4
Print Round(4.45, 1)                    ' Prints 4.4

' Round to the nearest hundred.
Print Round(153.33, -2)                 ' Prints 200
```

RSet statement

Assigns a specified string to a string variable and right-aligns the string in the variable.

Syntax

RSet *stringVar* **=** *stringExpr*

Elements

stringVar

The name of a fixed-length String variable, a variable-length String variable, or a Variant variable.

stringExpr

The string to be assigned to the variable and right-aligned.

Usage

If the length of *stringVar* is greater than the length of *stringExpr*, LotusScript right-aligns *stringExpr* within *stringVar* and sets the remaining characters in *stringVar* to spaces.

If the length of *stringVar* is less than the length of *stringExpr*, LotusScript copies only as many left-most characters from *stringExpr* as will fit within *stringVar*.

If *stringVar* contains a numeric value, LotusScript converts it to String to determine the length of the result.

If *stringVar* is a Variant, it can't contain NULL.

You cannot use RSet to assign variables of one user-defined data type to variables of another user-defined data type.

Examples: RSet statement

Example 1

```
Dim positFin As String * 20       ' String of 20 null characters
RSet positFin$ = "Right"          ' "Right" is shorter than positFin.
Print positFin$                   ' Prints "               Right"
' The string "Right" is right-aligned in the fixed-length String
' variable named positFin, and the initial 15 characters in positFin
' are set to spaces.
```

Example 2

```
Dim x As Variant
x = "q"
RSet x = "ab"
Print x                          ' Prints "a"
' The string "q" is assigned to the Variant variable x, giving it a
' length of 1. The single leftmost character "a" of the two-character
' string expression "ab" is assigned to x.
```

RTrim function

Remove trailing spaces from a string and return the resulting string.

Syntax

RTrim[**$**] **(** *stringExpr* **)**

Elements

stringExpr

Any String expression.

Return value

RTrim returns a Variant of DataType 8 (String), and RTrim$ returns a String. RTrim returns the trimmed version of *stringExpr*, but does not modify the contents of *stringExpr* itself.

Examples: RTrim function

```
Dim trimRight As String
trimRight$ = RTrim$("   abc  ")
Print trimRight$
Print Len(trimRight$)
' Output:
'    abc
'  6
' The string "   abc" is assigned to trimRight.
' Note that the leading spaces were not removed.
```

Run statement

LotusScript Release 3 no longer supports the Run statement. To execute a Lotus product macro, use the Evaluate function or statement.

Second function

Returns the second of the minute (an integer from 0 to 59) for a date/time argument.

Syntax

Second (*dateExpr* **)**

Elements

dateExpr

Any of the following kinds of expression:

- A valid date/time string of String or Variant data type. In a date/time string, a 2-digit designation of a year is interpreted as that year in the twentieth century. For example, 17 and 1917 are equivalent year designations.
- A numeric expression whose value is a Variant of DataType 7 (Date/Time).
- A number within the valid date range: the range -657434 (representing Jan 1, 100 AD) to 2958465 (Dec 31, 9999 AD).
- NULL.

Return value

Second returns an integer between 0 and 59.

The data type of Second's return value is a Variant of DataType 2 (Integer).

Second(NULL) returns NULL.

Examples: Second function

```
' Construct a message that displays the current time and
' the number of hours, minutes, and seconds remaining in the day.
Dim timeFrag As String, hoursFrag As String
Dim minutesFrag As String, secondsFrag As String
Dim crlf As String, message As String
timeFrag$ = Format(Time, "h:mm:ss AM/PM")
hoursFrag$ = Str(23 - Hour(Time))
minutesFrag$ = Str(59 - Minute(Time))
secondsFrag$ = Str(60 - Second(Time))
crlf$ = Chr(13) & Chr(10)               ' Carriage return/line feed
message$ = "Current time: " & timeFrag$ & ". " & crlf$ _
   & "Time remaining in the day: " _
   & hoursFrag$ & " hours, " _
   & minutesFrag$ & " minutes, and " _
   & secondsFrag$ & " seconds."
MessageBox(message$)
```

Seek function

Returns the file position (the byte position in a binary file or the record number in a random file) in an open file.

Syntax

Seek (*fileNumber* **)**

Elements

fileNumber

The number assigned to the file when it was opened with the Open statement.

Return value

Seek returns a Long value between 1 and 2.0E31 - 1, inclusive, unless the file position is very large. For a file position larger than 2.0E30, the return value is negative.

For a binary or sequential file, Seek returns the current byte position within the file.

For a random file, Seek returns the number of the next record within the file.

Usage

The first byte or record in a file is always file position 1.

Examples: Seek function

```
Type personRecord
   empNumber As Integer
   empName As String * 20
End Type

Dim rec1 As personRecord, rec2 As personRecord
Dim fileNum As Integer, recNum As Integer
Dim fileName As String
fileNum% = FreeFile()
fileName$ = "data.txt"
recNum% = 5

Open fileName$ For Random As fileNum% Len = Len(rec1)
rec1.empNumber% = 123
rec1.empName$ = "John Smith"
Print Seek(fileNum%)          ' Prints 1 for current position
Put #fileNum%, recNum%, rec1  ' Write data at record 5
Print Seek(fileNum%)          ' Prints 6

Seek fileNum%, 1              ' Rewind to record 1
Print Seek(fileNum%)          ' Prints 1
Rec2.empNumber% = 456
Rec2.empName$ = "Jane Doe"
Put #fileNum%, , rec2         ' Write at current position
Print Seek(fileNum%)          ' Prints 2

Close fileNum%
```

Seek statement

Sets the file position (the byte position in a binary file or the record number in a random file) in an open file.

Syntax

Seek [**#**]*fileNumber* **,** *position*

Elements

fileNumber

The number assigned to the file when it was opened with the Open statement.

position

The desired file position for the next read or write operation. In a binary or sequential file, this is a non-zero byte location; in a random file, this is a record number (in a random file).

In a binary or sequential file, the first byte is byte number 1; in a random file, the first record is record number 1.

If *position* is zero or is omitted, Seek returns an error.

Usage

The record number in a Get statement or a Put statement overrides a file position set by a Seek statement.

Writing to a file after moving the file position beyond the end of the file appends data to the end of the file.

Examples: Seek statement

```
Type personRecord
   empNumber As Integer
   empName As String * 20
End Type

Dim rec1 As personRecord, rec2 As personRecord
Dim fileNum As Integer, recNum As Integer
Dim fileName As String
fileNum% = FreeFile()
fileName$ = "data.txt"
recNum% = 5

Open fileName$ For Random As fileNum% Len = Len(rec1)
rec1.empNumber% = 123
rec1.empName$ = "John Smith"
Print Seek(fileNum%)          ' Prints 1 for current position
Put #fileNum%, recNum%, rec1  ' Write data at record 5
Print Seek(fileNum%)          ' Prints 6

Seek fileNum%, 1              ' Rewind to record 1
Print Seek(fileNum%)          ' Prints 1
```

```
Rec2.empNumber% = 456
Rec2.empName$ = "Jane Doe"
Put #fileNum%, , rec2          ' Write at current position
Print Seek(fileNum%)           ' Prints 2

Close fileNum%
```

Select Case statement

Selects a group of statements to execute, based on the value of an expression.

Syntax

Select Case *selectExpr*

[**Case** *condList*

[*statements*]]

[**Case** *condList*

[*statements*]]

...

[**Case Else**

[*statements*]]

End Select

Elements

selectExpr

An expression whose value is compared with values in the subsequent *condList* conditions. This expression is evaluated once, and its value is used repeatedly for comparison.

condList

Each *condList* is a list of conditions, one of which must be met for the subsequent group of statements to execute. Each condition takes one of the forms listed below, where *expr* is any expression:

- *expr*

  Returns TRUE if *selectExpr* matches *expr* exactly.

- *expr* **To** *expr*

  Returns TRUE if the *selectExpr* falls inclusively within this range.

  For example, if you specify 25 To 50, the corresponding group of statements is executed when *selectExpr* is any value between 25 and 50, inclusive.

- **Is** *comparisonOp expr*

  Returns TRUE when the comparison operation for *selectExpr* and *expr* is true. The comparison operator must be one of the following: `= > < <> >< <= =< >= =>`.

  For example, if you specify Is < 37, then the corresponding group of statements is executed when *selectExpr* is less than 37.

statements

Statements to be executed if one of the governing conditions in the associated *condList* is the first condition to be satisfied.

Usage

The *selectExpr* is compared against each condition, within each *condList* in succession. The first time that a condition in some *condList* is satisfied, the group of statements associated with that *condList* is executed and the selection operation ends.

Either a single group of *statements* is executed, or no statements are executed. If you include a Case Else group of statements, it's executed only if *selectExpr* fails all conditions in all c*ondList* arguments.

Examples: Select Case statement

```
' One of five Print statements is selected for execution,
' depending on the value of the variable segSelect.
' Note that the Case Else clause is executed only if segSelect
' is less than 0, between 0 and 1, between 1 and 2, between 2 and 3,
' or between 5 and 6.

Dim segSelect As Double
' ...
For segSelect# = -1 to 7
   Select Case segSelect#
      Case 0       : Print "0"
      Case 1, 2    : Print "1, 2"
      Case 3 To 5  : Print "3 TO 5"
      Case Is >= 6 : Print ">=6"
      Case Else    : Print "Else"
   End Select
Next
' Output:
' Else
' 0
' 1, 2
' 1, 2
' 3 TO 5
' 3 TO 5
' 3 TO 5
' >=6
' >=6
```

SendKeys statement

Enters keystrokes in the active window as if they were entered from the keyboard.

Syntax

SendKeys *string* [**,** *processNow*]

string

Any string expression, specifying a sequence of keystrokes to be sent to the active window.

To repeat a keystroke in *string*, use the code {*key count*}, where *key* is the keystroke to repeat, and *count* is the number of times to repeat it. For example, "{RIGHT 3}" represents pressing the Right Arrow key three times.

Include a space between *key* and *count*; otherwise {*key count*} may be interpreted as a function key specification. For example, "{F 4}" represents pressing the letter F four times, but "{F4}" represents pressing the function key F4.

processNow

Optional. Any numeric value. A nonzero value is interpreted as TRUE; a zero (0) is interpreted as FALSE.

- If *processNow* is TRUE, script execution does not continue until after all characters in *string* have been processed by the active window.
- If *processNow* is FALSE, script execution continues immediately, whether or not *string* has been fully processed.

The default value of *processNow* is FALSE. You will usually want to specify TRUE for *processNow*.

Usage

The SendKeys statement is not legal at the module level.

SendKeys is not supported on Macintosh and UNIX platforms.

To send an ordinary keyboard key or sequence of keys, such as A or 8 or DIR, simply include the character(s) in *string*.

To send non-printing keyboard keys, such as Tab or Backspace, or keys that perform actions in the active window, such as Page Up, use the key code from the following table in *string*.

Key	*Code*
Backspace	{BS} or {BKSP} or {BACKSPACE}
Break	{BREAK}
Caps Lock	{CAPSLOCK}
Clear	{CLEAR}
Del	{DEL} or {DELETE}
Down arrow	{DOWN}
End	{END}
Enter	~ or {ENTER}
Esc	{ESC} or {ESCAPE}
Help	{HELP}
Home	{HOME}
Ins	{INSERT}
Left arrow	{LEFT}
Num Lock	{NUMLOCK}
Pg Dn	{PGDN}
Pg Up	{PGUP}
Right arrow	{RIGHT}
Scoll Lock	{SCROLLLOCK}
Tab	{TAB}
Up arrow	{UP}
Function keys	{F1} to {F16}

To include a character from the following table in *string*, enclose it in braces as shown.

Character	*Code*
Brace	{**{**} *or* {**}**}
Bracket	{**[**} *or* {**]**}
Caret	{^}
Parenthesis	{**(**} *or* {**)**}
Percent sign	{**%**}
Plus sign	{**+**}
Tilde	{**~**}

The following table shows how to designate keys pressed in combination with Alt, Ctrl, or Shift.

Combination key	*Code*	*Example*
Alt	**%**	%{F4} represents Alt+F4
Ctrl	**^**	^{F4} represents Ctrl+F4
Shift	**+**	+{F4} represents Shift+F4

To apply a combination key to a sequence of keys, enclose the sequence in parentheses. For example, +(xy) holds down the Shift key for both x and y. It is equivalent to +x+y.

SendKeys cannot send keystrokes to a window that is not a Windows program, and cannot send the Print Scrn key to any program.

SendKeys generates an "Illegal function call" error if *string* contains any of the following:

- An unmatched parenthesis
- An illegal key code
- An illegal repeat count
- Too many characters

Note that SendKeys is often useful after Shell, to send keystrokes to the program that Shell started. Remember that Shell does not guarantee that the program is loaded before executing the statements that follow it.

Examples: SendKeys statement

```
' Use Shell to open the Windows Notepad. Then use SendKeys to send
' a note entered by the user to Notepad. The user can continue
' composing the note and use Notepad to save it as a text file.
Sub WriteNote
   Dim taskId As Integer, note As String
   note$ = InputBox("Start your note:")
   taskId% = Shell("notepad.exe", 1)
   SendKeys note$, TRUE
End Sub
WriteNote               ' Call the WriteNote sub.
```

Set statement

Assigns an object reference to a variable, or associates an object with a variable.

Use one of the following three syntaxes:

Syntax 1: Create an object and assign a reference

Set *var* **=** **New** *class* [**(** [*argList*] **)**]

Elements

var

A Variant variable, an object of the class *class*, an object of a class derived from *class*, or any variable element that accepts an object reference, such as an element of an array, list, or user-defined data type.

class

The name of the user-defined or product class of the object to be created.

argList

For user-defined classes, *argList* is the comma-separated list of arguments required by the class constructor sub New, defined in the class named by *type*. For product classes, consult the product documentation.

Syntax 2: Copy an existing object reference to another variable

Set *var1* **=** *var2*

Elements

var1

A Variant variable, an object of the same class as *var2*, an object of a class derived from *var2*'s class, or any variable element that accepts an object reference, such as an element of an array, list, or user-defined data type.

var2

An expression whose value is NOTHING, an object reference of the same class as *var1*, an object reference of a class derived from *var1*'s class, or an object reference of a class from which *var1* is derived. In the latter case, *var2* must contain an instance of *var1*'s class or a class derived from *var1*.

Syntax 3: Associate a product object with a variable

Set *var* **=** **Bind** [*prodClass*] **(** *objectName* **)**

Elements

var

A Variant variable, an object of *prodClass*, or any variable element that accepts an object reference, such as an element of an array, list, or user-defined data type.

Bind

The Bind keyword associates *objectName* with *var*. The association is made by name, and is valid until any of the following conditions is true:

- *var* is out of scope.
- *objectName* no longer exists.
- *var* is set to another value.

prodClass

Optional. The product class of the object *objectName*. If *prodClass* is not specified, LotusScript assumes that *objectName* is of the same class as *var*. If *var* is a Variant, you must include *prodClass.*

objectName

A string specifying the name and, optionally, the path of the product object of class *prodClass.*

The form of this string is product-specific. For example, the product object name might have the form *"ApplicationWindowName\ObjectName."* Refer to your Lotus product documentation for information about specifying product object names.

Usage

The Set statement is the object reference assignment statement. It is parallel to the Let statement, the general assignment statement for variables of all types except object reference variables.

When you use the user interface, rather than a script, to create a product object, some Lotus products implicitly declare the name you (or the product) have assigned the object as an object reference variable and bind it to the object. This allows you to use the object name in scripts without explicitly declaring a variable and binding it to the object.

To test an object reference variable for the NOTHING value, use the Is operator.

Examples: Set statement

Example 1 (Syntax 1)

```
' The variable terPoint is declared as an object reference variable of
' the class Point, which must already be defined. The New sub for
' class Point has no arguments. The Set statement creates a new object
' of the class Point and assigns its reference to terPoint.
Dim terPoint As Point
Set terPoint = New Point
```

Example 2 (Syntax 2)

```
' The class Worker and the class Carpenter must already be defined,
' with Carpenter as a derived class of Worker. The first Dim statement
' declares x as an object reference variable of the class Worker.
' The second Dim statement declares y as an object reference variable
' of the class Carpenter. This statement also creates a new object of
' the class Carpenter, named "Terry"; and assigns its reference to the
' object reference variable y. The Set statement assigns the reference
' in y to the object reference variable x. (A reference to a Carpenter
' can be assigned to a variable of class Worker because Worker is the
' base class of Carpenter.)
Dim x As Worker
Dim y As New Carpenter("Terry")
Set x = y
```

Example 3 (Syntax 3)

```
' The Dim statement declares icCheckBox as an object reference
' variable of the pre-defined product class Check. The Set statement
' binds the object reference variable icCheckBox to
' the product object Checkbox1.
Dim icCheckBox As Check
Set icCheckBox = Bind("Checkbox1")
```

SetFileAttr statement

Sets the system attributes of a file.

Syntax

SetFileAttr *fileName* **,** *attributes*

SetAttr is acceptable in place of SetFileAttr.

Elements

fileName

A string expression; you can optionally include a path.

attributes

The attributes to apply to the file, expressed as the sum of any of the following Integer values:

Value	*Description*	*Constant*
0	Normal file	ATTR_NORMAL
1	Read-only	ATTR_READONLY
2	Hidden	ATTR_HIDDEN
4	System	ATTR_SYSTEM
32	Changed since last back-up	ATTR_ARCHIVE

The constants are defined in the file lsconst.lss. Including this file in your script allows you to use constant names instead of the corresponding numeric values.

Usage

Do not use SetFileAttr on an open file, unless the file has been opened as read-only.

Examples: SetFileAttr statement

```
' This script creates a file and uses SetFileAttr to set the file
' attributes to Read-Only, System, and Hidden. It then uses
' GetFileAttr to verify the file attributes.
%Include "lsconst.lss"
Dim fileNum As Integer, attr As Integer
Dim fileName As String, msg As String
fileNum% = FreeFile()
fileName$ = "data.txt"

Open fileName$ For Output As fileNum%
Close fileNum%

SetFileAttr fileName$, ATTR_READONLY + ATTR_SYSTEM + ATTR_HIDDEN
attr% = GetFileAttr(fileName$)
If (attr% And ATTR_READONLY) Then
   msg$ = msg$ & " Read-Only "
Else
   msg$ = msg$ & " Normal "
End If
If (attr% And ATTR_HIDDEN)    Then msg$ = msg$ & " Hidden "
If (attr% And ATTR_SYSTEM)    Then msg$ = msg$ & " System "
If (attr% And ATTR_VOLUME)    Then msg$ = msg$ & " Volume "
If (attr% And ATTR_DIRECTORY) Then msg$ = msg$ & " Directory "
Print msg$

SetFileAttr fileName$, ATTR_NORMAL   ' Reset to normal
Kill fileName$
```

Sgn function

Identifies the sign (positive or negative) of a number.

Syntax

Sgn (*numExpr* **)**

Elements

numExpr

Any numeric expression.

Return value

The following table shows the values that the Sgn function returns.

Sign of numExpr	*Value*
Negative	-1
Zero	0
Positive	1

Examples: Sgn function

```
Dim x As Integer, y As Integer
x% = Sgn(-45)
Print x%                          ' Prints -1
y% = Sgn(12)
Print y%                          ' Prints 1
Print Sgn(x% + y%)                ' Prints 0
```

Shell function

Starts another program.

Syntax

Shell (*program* [**,** *windowStyle*] **)**

Elements

program

A string expression whose value is the name of the program to run, including arguments. *program* is the name of an executable file that uses a file name extension of BAT, COM, PIF, or EXE. You can omit the file name extension, and you can optionally include a complete path specification.

Using an internal DOS command name generates an error.

windowStyle

Optional. A number designating a valid window style, as specified in the following table.

Style	*Description*	*Constant*
1, 5, or 9	Normal with focus	SHELL_NORMAL_FOCUS
2	Minimized with focus (default)	SHELL_MIN_FOCUS
3	Maximized with focus	SHELL_MAX_FOCUS
4 or 8	Normal without focus	SHELL_NORMAL_NO_FOCUS
6 or 7	Minimized without focus	SHELL_MIN_NO_FOCUS

The constants are defined in the file lsconst.lss. Including this file in your script allows you to use constant names instead of the numeric values assigned to them.

Return value

If the operating system is Windows 3.1 and LotusScript successfully starts *program*, Shell returns the program's task ID, a number that uniquely identifies the program. If the operating system is Windows NT and LotusScript successfully starts *program*, Shell returns the number 33.

If LotusScript cannot start *program*, Shell returns an error.

Usage

Shell must be called from within an expression or an assignment statement, so that its return value is used.

After Shell starts a program, LotusScript continues executing the script without waiting to make sure the program has completed. You cannot be sure that a program started by Shell has finished running before the rest of your script is executed.

Examples: Shell function

```
' Start the Windows Calculator as a normal (not minimized)
' window with focus.
Dim taskId As Integer
taskId% = Shell("CALC.EXE", 1)
```

Sin function

Returns the sine, in radians, of an angle.

Syntax

Sin (*angle* **)**

Elements

angle

Any numeric expression. It is interpreted as an angle expressed in radians.

Return value

Sin returns the sine of *angle*, a Double between -1 and 1, inclusive.

Examples: Sin function

```
' Convert the angle of 45 degrees to radians,
' then compute and print the sine of that angle.
Dim degrees As Double, radians As Double
degrees# = 45
radians# = degrees# * (PI / 180)
Print Sin(radians#)          ' Prints .707106781186548
```

Single data type

Specifies a variable that contains a 4-byte floating-point value.

Usage

The Single suffix character for implicit data type declaration is the exclamation point (!).

Single variables are initialized to zero (0).

The range of Single values is -3.402823E+38 to 3.402823E+38, inclusive.

The smallest nonzero Single value, disregarding sign, is 1.175494351E-38.

LotusScript aligns Single data on a 4-byte boundary. In user-defined data types, declaring variables in order from highest to lowest alignment boundaries makes the most efficient use of data storage space.

Examples: Single data type

```
' Explicitly declare a Single variable.
Dim x As Single

' Implicitly declare a Single variable.
mole! = 6.02E23

Print mole!   ' Prints the value of mole.
```

Space function

Returns a specified number of spaces as a string.

Syntax

Space[**$**] **(** *numExpr* **)**

Elements

numExpr

Any numeric expression. If *numExpr* includes a fractional part, LotusScript rounds it to the nearest integer.

Return value

The return value contains *numExpr* space characters. Space returns a Variant of DataType 8 (String), and Space$ returns a String.

Examples: Space function

```
' Assign a string of four spaces to the variable smallTab.
Dim smallTab As String
smallTab$ = Space$(4)
Print Len(smallTab$)
' Output:
' 4
```

Spc function

Inserts a specified number of spaces in the output from a Print or Print # statement, beginning at the current character position.

Syntax

Spc (*numExpr* **)**

Elements

numExpr

Any numeric expression whose value is between 0 and 32000, inclusive. *numExpr* designates the number of spaces to insert in the Print output.

Usage

If you specify a width for the file (you can set the width only for printed files), *numExpr* interacts with that width as follows:

- If *numExpr* is smaller than the width, LotusScript prints *numExpr* spaces.
- If *numExpr* is larger than the width, LotusScript prints as many spaces as fit on one line, with the remainder appearing on the next line, until *numExpr* spaces have been printed.

If you don't specify a width for the file, LotusScript prints exactly *numExpr* spaces.

Examples: Spc function

```
' The Print # statement prints numbers with a leading space (omitted
' if the number is negative) and a trailing space.

' In this example, Spc(1) inserts another space following each number
' and its trailing space. The second and fourth lines each begin with
' two spaces: the first space on the line is generated by Spc(1), and
' the second space on the line is the leading space before the number
' first printed on the line (3 or 8).

' In the second line, the number 4 is followed by three spaces.
' These last four characters can be read as
' "4, trailing space, Spc(1), leading space".

Open "spc.tst" For Output As #1
' Define line width in SPC.TST as 10 characters.
```

```
Width #1, 10
For i = 0 To 9
   Print #1, i; Spc(1);
Next i
Close #1
' Output to the file (the display of each line here includes
' a leading quote character (') and a leading space):
'  0   1   2
'   3   4
' 5   6   7
'   8   9
```

Sqr function

Returns the square root of a number.

Syntax

Sqr (*numExpr* **)**

Elements

numExpr

Any numeric expression greater than or equal to zero.

Return value

Sqr returns a Double. If *numExpr* is negative, Sqr returns an error.

Examples: Sqr function

```
Dim root As Double
root# = Sqr(169)
Print root#                      '  Prints 13
```

Stop statement

Simulates the occurrence of a breakpoint.

Syntax

Stop

Usage

The Stop statement suspends execution of the script and transfers control to the LotusScript debugger as though a breakpoint is set at the Stop statement.

The Stop statement is legal within a procedure or class. It is not legal at the module level.

Str function

Returns the String representation of a number.

Syntax

Str[**$**] **(** *numExpr* **)**

Elements

numExpr

Any numeric expression.

Return value

Str returns a Variant of DataType 8 (a string), and Str$ returns a String.

Usage

When LotusScript represents a positive number as a String, it inserts a leading space.

Examples: Str function

```
' Assign the strings " 123" and "-123" to the variables string1
' and string2, respectively.
' For the positive value, note the addition of a leading space.
Dim string1 As String, string2 As String
string1$ = Str$(123)                    ' Assigns " 123"
string2$ = Str$(-123)                   ' Assigns "-123"
Print string2$; string1$
' Output:
' -123 123
```

StrCompare function

Compares two strings and returns the result.

Syntax

StrCompare (*string1* **,** *string2* [**,** *compMethod*] **)**

StrComp is acceptable in place of StrCompare.

Elements

string1

Any String expression.

string2

Any String expression.

compMethod

A number designating the comparison method. Use 0 for case-sensitive and pitch-sensitive, 1 for case-insensitive and pitch-sensitive, 4 for case-sensitive and pitch-insensitive, 5 for case-insensitive and pitch-insensitive. Use 2 to specify string comparison in the platform's collation sequence. If 2 is specified, strings are compared bit-wise. If you omit *compMethod*, the default comparison mode is the mode set by the Option Compare statement for this module. If there is no statement for the module, the default is case-sensitive and pitch-sensitive.

Return value

The following table shows what StrCompare returns, depending on the relationship between the strings being compared.

Strings being compared	*StrCompare result*
Either string is NULL	NULL
string1 is less than *string2*	-1
string1 equals *string2*	0
string1 is greater than *string2*	1

Examples: StrCompare function

```
' The following results are for LotusScript in English,
' running on Windows 3.1.
Print StrCompare("abc", "ab", 0)   ' Prints 1
Print StrCompare("ab", "abc", 0)   ' Prints -1
Print StrCompare("AB", "ab", 1)    ' Prints 0
Print StrCompare("AB", "ab", 2)    ' Prints -1
```

StrConv function

Converts a string to a different case or character set.

Syntax

StrConv (*expr* **,** *conversionType* **)**

Elements

expr

A string or numeric expression. A numeric expression is converted to a string.

conversionType

An integer that defines the type of conversion:

Constant name	*Value*	*Type of conversion*
SC_UpperCase	1	Uppercase
SC_LowerCase	2	Lowercase
SC_ProperCase	3	Proper case
SC_Wide	4	Single byte to double byte
SC_Narrow	8	Double byte to single byte
SC_Katakana	16	Hiragana to Katakana
SC_Hiragana	32	Katakana to Hiragana

Return value

The return value is a variant containing the result of the conversion.

Usage

The valid values for the *conversionType* elements listed in the preceding table are defined as constants in the file lsconst.lss. If you want to use the constants instead of numbers, include this file in your script.

ConversionType values can be combined (*or*ed) as follows:

- Any combination of SC_UpperCase, SC_LowerCase, and SC_ProperCase causes SC_ProperCase.
- Combining SC_Wide and SC_Narrow is illegal.
- Combining SC_Katakana and SC_Hiragana is illegal.
- If combined, the following operations occur in the following order: case operation, SC_Wide, SC_Katakana. Case operations are applied to double-byte alphanumeric characters.

If *expr* is the null string, the result is the null string. If *expr* is Null, the result is Null.

For proper case, the following numeric character codes are treated as word separators in a string literal: 0 (null), 9 (horizontal tab), 12 (form feed), 32 (space), 0x3000 (doubl-byte space). The following are treated as separators in a multi-line string: 10 (line feed), 13 (carriage return).

Examples: StrCompare function

```
%INCLUDE "lsconst.lss"
nameString$ = Inputbox$("Name?")
nameProper$ = Strconv(nameString$, SC_ProperCase)
Messagebox "nameProper = " & nameProper$
```

String data type

Specifies a variable used to store text strings, using the character set of the Lotus product that started LotusScript.

Usage

The String suffix character for implicit data type declaration is the dollar sign ($).

The declaration of a string variable uses this syntax:

Dim *varName* **As String** [***** *num*]

The optional *num* argument specifies that *varName* is a fixed-length string variable of *num* characters. A fixed-length string variable is initialized to a string of null characters (the character Chr(0)).

When you assign a string to a fixed-length string variable, LotusScript truncates a longer string to fit into the declared length. It pads a shorter string to the declared length with trailing spaces.

Fixed-length strings are often used in declaring data structures for use in file I/O or C access.

An implicitly declared String variable is always a variable-length string variable.

Variable-length strings are initialized to the empty string (“”).

LotusScript aligns variable-length String data on a 4-byte boundary. In user-defined data types, declaring variables in order from highest to lowest alignment boundaries makes the most efficient use of data storage space. Fixed-length strings are not aligned on any boundary.

Examples: String data type

```
' In this example, the variable-length String variable firstName and
' the fixed-length String variable homeState are explicitly declared
' and assigned appropriate String values. The variable adStreet is
' implicitly declared to be of type String by the $ suffix character.

' Explicitly declare a variable-length String variable.
Dim firstName As String
firstName$ = "Mark"

' Explicitly declare a fixed-length String variable.
Dim homeState As String * 4
homeState$ = "  MA"

' Implicitly declare a variable-length String variable.
adStreet$ = "123 Maple St."

Print firstName$   ' Prints "Mark"
Print adStreet$; homeState$   ' Prints "123 Maple St. MA"
```

String function

Returns a string consisting of a particular character, which is identified either by its platform-specific numeric character code, or as the first character in a string argument.

Syntax

String[**$**] **(** *stringLen* **,** { *charCode* | *stringExpr* } **)**

Elements

stringLen

A numeric expression whose value is the number of characters to put in the returned string. LotusScript rounds *stringLen* to the nearest integer.

charCode

A numeric expression of data type Long whose value specifies the platform-specific character code for each character in the string. The range of legal codes is platform-dependent.

stringExpr

Any string expression. The first character in this string is the character to be used in the returned string.

Return value

String returns a Variant of DataType 8 (String), and String$ returns a String.

Examples: String function

```
Dim stars As String, moreStars As String
stars$ = String$(4, Asc("*"))
moreStars$ = String$(8, "* characters")
Print stars$, moreStars$   ' Prints ****      ********
```

Sub statement

Defines a sub.

Syntax

[**Static**] [**Public** | **Private**] **Sub** *subName* [**(** [*argList*] **)**]

[*statements*]

End Sub

Elements

Static

Optional. Directs LotusScript to save the values of the sub's local variables between calls to the sub.

Public | **Private**

Optional. Public specifies that the sub is visible outside the scope (module or class) where the sub is defined, as long as this module is loaded. Private specifies that the sub is visible only within the current scope.

A sub in module scope is Private by default; a sub in class scope is Public by default.

subName

The sub name. The names Delete, Initialize, New, and Terminated are specialized. Use these names only as described in the topics Sub Delete, Sub Initialize, Sub New, and Sub Terminate.

argList

Optional. A comma-separated list of declarations for arguments to be passed to this sub when it is called.

The syntax for each argument declaration is:

ByVal *argument* [**( )** | **List**] [**As** *dataType*]

ByVal specifies that *argument* is passed by value: that is, the value assigned to *argument* is a copy of the value specified in the sub call, rather than a reference to the original value.

*argument***()** is an array variable. *argument* **List** identifies *argument* as a list variable. Otherwise, *argument* can be a variable of any of the other data types that LotusScript supports.

As *dataType* specifies the variable's data type. You can omit this clause and use a data type suffix character to declare the variable as one of the scalar data types. If you omit this clause and *argument* doesn't end in a data type suffix character (and isn't covered by an existing Def*type* statement), LotusScript assigns it the Variant data type.

Enclose the entire list of argument declarations in parentheses.

Usage

The Public keyword cannot be used in a product object script or %Include file in a product object script, except to declare class members. You must put such Public declarations in (Globals).

Arrays, lists, type instances, and objects can't be passed by value as arguments. They must be passed by reference.

A sub does not return a value.

A sub can be called in either of these two forms:

subName arg1, arg2, ...

Call *subName* (*arg1, arg2, ...*)

A sub definition can't contain the definition of another procedure (a function, sub, or property).

A sub member of a class cannot be declared Static.

You can exit a sub using an Exit Sub statement.

Your Lotus product can provide special named subs for use in your scripts; see the product documentation for more information.

Examples: Sub statement

Use a sub and a function to compute the cost of buying a house as follows.

- Ask the user for the price of the house, and call the ComputeMortgageCosts sub with *price* as the argument.
- The ComputeMortgageCosts sub gathers down payment (at least 10% of cost), annual interest rate, and the term of the mortgage from the user, then calls the Payment function with 3 arguments. Annual interest and term (years) are passed by value rather than reference, so the Payment function can adjust them to compute monthly rate and monthly payment without changing the values of these variables in the ComputeMortgageCosts sub.
- If the user enters positive values, Payment returns the monthly payment. Otherwise, it returns 0. ComputeMortgageCosts then constructs an appropriate message.

```
Dim price As Single, message As String

Function Payment (princpl As Single, _
                  ByVal intrst As Single, _
                  ByVal term As Integer) As Single
   intrst! = intrst!/12
   term% = term% * 12
   ' If any of the parameters is invalid, exit the function
   ' (Payment will return the value 0).
   If princpl! <= 0 Or intrst! <= 0 Or term% < 1 Then _
      Exit Function
   ' The standard formula for computing the amount of the
   ' periodic payment of a loan:
   Payment = princpl! * intrst! /(1 - (intrst! + 1) ^ (-term%))
End Function

Sub ComputeMortgageCosts (price As Single)
   Dim totalCost As Single, downpmt As Single
   Dim mortgage As Single, intrst As Single
   Dim monthlypmt As Single, years As Integer
EnterInfo:
   downpmt! = CSng(InputBox("How much is the down payment?"))
   ' The downpayment must be at least 10% of the price.
   If downpmt! < (0.1 * price!) Then
```

```
      MessageBox "Your down payment must be at least " _
         & Format(price! * .1, "Currency")
      GoTo EnterInfo:
   End If
   mortgage! = price! - downpmt!
   intrst! = CSng(InputBox("What is the interest rate?"))
   years% = CInt(InputBox("How many years?"))
   ' Call the Payment function, which returns the monthly payment.
   monthlypmt! = Payment(mortgage!, intrst!, years%)
   totalCost! = downpmt! + (monthlypmt! * years% * 12)
   If monthlypmt! > 0 Then      ' Create a multiline message.
      message$ = _
|Price | & Format(price!, "Currency") & |
Down Payment: | & Format(downpmt!, "Currency") & |
Mortgage: | & Format(mortgage!, "Currency") & |
Interest: | & Format(intrst!, "Percent") & |
Term: | & Str(years%) & | years
Monthly Payment: | & Format(monthlypmt!, "Currency") & |
Total Cost: | & Format(monthlypmt! * years% * 12, "Currency")
   Else
      message$ = "You did not enter valid input."
   End If
End Sub

' Start here.
price! = CSng(InputBox("How much does the house cost?"))
' Call the Compute MortgageCosts sub.
ComputeMortgageCosts (price!)
' Display the message.
MessageBox message$
```

Sub Delete

A user-defined sub that LotusScript executes when you delete an object belonging to the class for which the Delete sub is defined.

Syntax

Sub Delete

[*statements*]

End Sub

Usage

In the definition for a user-defined class, you can define a destructor named Delete. This sub is automatically executed whenever you delete an object belonging to the class for which you defined the Delete sub.

The Delete sub is always Public: you can't declare it as Private.

The Delete sub can't take any arguments.

The Delete sub can't be called directly; it's invoked only when the object is deleted. The name Delete can only be used as the name of a destructor; it can't be used to name any other procedure or a variable, for example.

Examples: Sub Delete

```
' Define the class Customer.
Class Customer
   Public Name As String
   Public Address As String
   Public Balance As Currency

   ' Define a constructor sub for the class.
   Sub New (Na As String, Addr As String, Bal As Currency)
      Me.Name$ = Na$
      Me.Address$ = Addr$
      Me.Balance@ = Bal@
   End Sub

   ' Define a destructor sub for the class.
   Sub Delete
      Print "Deleting customer record for: "; Me.Name$
   End Sub
End Class

' Create an object of the Customer class.
Dim X As New Customer("Acme Corporation", _
   "55 Smith Avenue, Cambridge, MA", 14.92)
Print X.Balance@
' Output:
' 14.92

' Delete the object, first running the destructor sub.
Delete X
' Output:
' Deleting customer record for: Acme Corporation."

' Then the object is deleted.
```

Sub Initialize

A user-defined sub that LotusScript executes when the module containing the Initialize sub is loaded.

Syntax

Sub Initialize

[*statements*]

End Sub

Usage

Include in the Initialize sub any statements you want executed when LotusScript loads the containing module.

The Initialize sub is always Private.

The Initialize sub cannot take any arguments.

Examples: Sub Initialize

```
' When LotusScript loads the module, Initialize saves the name of the
' current working directory.
Dim StartDir As String
Sub Initialize          ' Store the current directory
   StartDir$ = CurDir$
End Sub

' The module changes the working directory.
' ...
' ...

' When LotusScript unloads the module, Terminate changes the working
' directory back to what it was when the module was loaded.
Sub Terminate ' Return to the startup directory.
   ChDir StartDir$
End Sub
```

Sub New

A user-defined sub that LotusScript executes when you create an object of the class for which the New sub is defined.

Syntax

Sub New [**(** [*argList*] **)**] [**,** *baseClass* **(** [*baseArgList*] **)**]

[*statements*]

End Sub

Elements

argList

Optional. A comma-separated list of parameter declarations for the New sub, enclosed in parentheses. Use the following syntax for each parameter declaration:

[**ByVal**] *paramName* [**( )** | **List**] [**As** *dataType*]

ByVal means that *paramName* is passed by value: that is, the value assigned to *paramName* is a copy of the value specified in the sub call, rather than a reference to the original value.

paramName() is an array variable; **List** identifies *paramName* as a list variable; otherwise, *paramName* can be a variable of any of the other data types that LotusScript supports.

As *dataType* specifies the variable data type. You can omit this clause and use a data type suffix character to declare the variable as one of the scalar data types. If you omit this clause and *paramName* doesn't end in a data type suffix character (and isn't covered by an existing Def*type* statement), its data type is Variant.

If the New sub for the derived class has no arguments, and the New sub for the base class has no arguments, omit (*argList*) and *baseClass* (*baseArgList*).

baseClass **(** [*baseArgList*] **)**

Optional. The *baseClass* is the name of the class from which the derived class is derived. This name must match the *baseClass* name in the Class statement for the derived class.

The *baseArgList* is a comma-separated list of arguments for the sub New of the base class. Note that these are actual arguments, not parameter declarations. This syntax enables a call of the New sub for the derived class to furnish actual arguments to the call of the New sub for the base class.

Include this syntax in the New sub only if all of these conditions are true:

- The class being defined is a derived class.
- The New sub for the base class of this derived class requires arguments.

Note that these arguments must be furnished to the New sub for the base class through the call of the New sub for the derived class.

- The argument list for the sub New of the base class does not match the argument list for the sub New of the derived class in number and data type of arguments; or you want to pass different arguments to the base class's sub New than those passed to the derived class's sub New.

When the class being defined is a derived class, each call of the New sub for the derived class generates a call of the New sub for the base class. If that base class is itself a derived class of another base class, another call is generated, and so on.

Usage

In the definition for a user-defined class, you can include a definition for the constructor sub, named New. If the definition exists, LotusScript calls this sub whenever it creates an object from that class. LotusScript calls the sub immediately after creating the object.

Examples: Sub New

```
' Define a class.
Class textObject

   ' Declare member variables.
   backGroundColor As Integer
   textColor As Integer
   contentString As String

   ' Define constructor sub.
   Sub New (bColor As Integer, tColor As Integer, cString As String)
      backGroundColor% = bColor%
      textColor% = tColor%
      contentString$ = cString$
      Print "Creating new instance of text object ..."
      Print "Text object state:"
      Print "Background color:" ; Me.backGroundColor% ; _
         "Text color:" ; Me.textColor%
   End Sub

   ' Define destructor sub.
   Sub Delete
      Print "Deleting text object."
   End Sub

   ' Define a sub to invert background and text colors.
   Sub InvertColors
      Dim x As Integer, y As Integer
      x% = backGroundColor%
      y% = textColor%
      Me.backGroundColor% = y%
```

```
      Me.textColor% = x%
   End Sub

End Class

' Create a new object of class textObject.
Dim zz As New textObject(0, 255, "This is my text")
' Output:
' Creating new instance of text object ...
' Text object state:
' Background color: 0 Text color: 255

' Invert the object's background and text colors.
zz.InvertColors
' Delete the object, first running the destructor sub.
Delete zz
' Output: Deleting text object.
```

Sub Terminate

A user-defined sub that LotusScript executes when the module containing the Terminate sub is unloaded.

Syntax

Sub Terminate

[*statements*]

End Sub

Usage

Include in the Terminate sub any statements you want executed when LotusScript unloads the containing module.

The Terminate sub is always Private.

The Terminate sub cannot take any arguments.

Examples: Sub Terminate

```
' When LotusScript loads the module, Initialize saves
' the name of the current working directory.
Dim startDir As String
Sub Initialize      ' Store the current directory.
   startDir$ = CurDir$
End Sub

' The module changes the working directory.
' ...
' ...
```

```
' When LotusScript unloads the module, Terminate changes the working
' directory back to what it was when the module was loaded.
Sub Terminate       ' Return to the startup directory.
   ChDir startDir$
End Sub
```

Tab function

Moves the print position to a specified character position within a line, when called from within a Print or Print # statement.

Syntax

Tab (*column* **)**

Elements

column

Any integer expression between 1 and 32000, inclusive, specifying a character position in the printed output. If *column* is less than 1, the Tab position defaults to 1 (the leftmost print position).

Usage

If you haven't specified a width for the file, Tab checks *column* against the current print position, and acts as follows:

- If you've already printed past the position specified by *column*, Tab prints a newline character, and then prints the next character in the *column* position on the next line.
- If *column* is at the current position, or after the current position, Tab prints enough spaces to move to the position specified by *column* and prints the next character in the *column* position on the current line.

If you print to a file whose width was set with the Width # statement, Tab interacts with that width as described in the following table.

Column	*Tab moves to:*
> *width*	*column* Mod *width*
< 1	column 1
< current print position	*(column* - current position) on the next line
> current print position	*(column* - current position) on the same line

Examples: Tab function

```
Dim firstN As String, lastN As String
firstN$ = "Bob"
lastN$ = "Jeremiah"
Print firstN$; Tab(5); lastN$; Tab(1); lastN$; Tab(2); lastN$; _
   Tab(3); lastN$
```

LotusScript prints the contents of firstN and lastN, using Tab() to separate them as follows:

```
Bob Jeremiah
Jeremiah
 Jeremiah
  Jeremiah
```

The semicolons in the Print statement are optional; they have no effect on the output, because the print position is determined by Tab.

Tan function

Returns the tangent, in radians, of an angle.

Syntax

Tan (*angle* **)**

Elements

angle

Any numeric expression. It is interpreted as an angle expressed in radians.

Return value

Tan returns a Double.

Examples: Tan function

```
' Convert the angle of 45 degrees to radians, and then
' compute and print the tangent of that angle.
Dim degrees As Double, radians As Double
degrees# = 45
radians# = degrees# * (PI / 180)
Print Tan(radians#)    ' Prints 1
```

Time function

Returns the system time as a time value.

Syntax
Time[**$**]

Return value
Time returns a time value representing the system time.

The return value is the fractional part of the value returned by the Now function. Time returns that value as a Variant of DataType 7 (Date / Time). Time$ returns that value as a String.

Both forms return the time rounded to the nearest second.

Usage
You can call the Time function as either Time or Time( ). You can call the Time$ function as either Time$ or Time$( ).

Examples: Time function
```
Dim current As String
current$ = Time$()
Print current$        ' Prints the system time
```

Time statement

Sets the system time to a specified time.

Syntax
Time[**$**] **=** *timeExpr*

Elements
timeExpr

Any expression whose value is a valid date / time value: either a String in a valid date / time format, or else a Variant containing either a date / time value or a string value in date / time format.

Examples: Time statement
```
' Set the system time to 6:20:15 PM using 24-hour notation.
Time = "18:20:15"
```

TimeNumber function

Returns a time value for a specified hour, minute, and second.

Syntax

TimeNumber (*hour* **,** *minute* **,** *second* **)**

TimeSerial is acceptable in place of TimeNumber.

Elements

hour

A numeric expression representing an hour (0 to 23, inclusive).

minute

A numeric expression representing a minute (0 to 59, inclusive).

second

A numeric expression representing a second (0 to 59, inclusive).

Return value

TimeNumber returns a Variant of DataType 7 (Date/Time). Its value represents time of day as a fraction of 24 hours, measured from midnight.

Usage

You can use expressions for *hour*, *minute*, and *second* to compute a time relative to another time. For example:

```
TimeNumber(3, 5, 5 - 10)
```

computes the time 10 seconds before 3:05:05 AM (the result is 3:04:55 AM).

Examples: TimeNumber function

```
' Print the time value for an hour, minute, and second.
Print TimeNumber(12, 30, 15)   ' Prints 12:30:15 PM
```

Timer function

Returns the time elapsed since midnight, in seconds.

Syntax

Timer

Return value

Timer returns the number of seconds elapsed since midnight as a Single value.

Usage

LotusScript rounds the number of seconds to the nearest hundredth.

The Randomize Statement uses the return value from Timer as its default seed value.

You can call the function as either Timer or Timer( ).

Examples: Timer function

```
' Calculate how long it takes the following loop to iterate
' 1000 times.
Dim startTime As Single
Dim elapsedTime As Single

startTime! = Timer()
For counter% = 1 To 10000
Next counter%
elapsedTime! = Timer() - startTime!

Print "10000 iterations in "; elapsedTime; " seconds"
```

TimeValue function

Returns the time value represented by a string expression.

Syntax

TimeValue (*stringExpr* **)**

Elements

stringExpr

A string expression that represents a valid date/time, or a Variant of DataType 7 (Date/Time). It can use either 12-hour or 24-hour format; for example, both "14:35" and "2:35PM" are valid. If you omit the seconds value in the *stringExpr* argument, it defaults to zero (0).

Return value

TimeValue returns a Variant of DataType 7 that contains a fractional date/time value.

Usage

If *stringExpr* specifies a date, TimeValue validates the date, but omits it from the return value.

Examples: TimeValue function

```
Dim fractionalDay As Single
fractionalDay! = TimeValue("06:00:00")
Print fractionalDay!
' Output:
' .25
' LotusScript assigns the value 0.25 to the variable fractionalDay,
' since 6:00 AM represents a time value of 6 hours, or one-quarter of
' a 24-hour day.
```

Today function

Returns the system date as a date value.

Syntax
Today

Return value
Today returns the system date as a Variant of DataType 7 (Date/Time).

The return value is the integer part of the value returned by the Now function.

Usage
The Today function is equivalent to the Date function.

You can call the function as either Today or Today( ).

Examples: Today function
```
' LotusScript assigns Today's date to the String variable whenNow.
Dim whenNow As String
whenNow$ = Today()
Print whenNow$
' Output:
' 6/7/95
```

Trim function

Removes leading and trailing spaces from a string and returns the resulting string.

Syntax
Trim[**$**] **(** *stringExpr* **)**

Elements
stringExpr
Any string expression.

Return value
Trim returns the trimmed version of *stringExpr*, but does not modify the contents of *stringExpr* itself.

Trim returns a Variant of DataType 8 (String), and Trim$ returns a String.

Examples: Trim function

```
Dim trimAll As String, testString As String
testString$ = "   a bc   "
' Trim the string, removing leading and trailing spaces.
' Embedded spaces are not removed.
trimAll$ = Trim$(testString$)   ' Assigns "a bc"
Print trimAll$
Print testString$               ' Unmodified by Trim()
' Output:
' a bc
'    a bc
```

Type statement

Defines a user-defined data type consisting of one or more members.

Syntax

[**Public** | **Private**] **Type** *typeName*

member declarations

End Type

Elements

Public | **Private**

Optional. Public specifies that the user-defined data type is visible outside the module where it is defined, as long as that module is loaded. Private specifies that the user-defined data type is visible only within the module where it is declared.

A type is Private by default.

typeName

The name of the type.

member declarations

Declarations for the members of the type. There must be at least one declaration in the type; the declarations cannot include Const statements.

Usage

Defining types

A Type statement is valid only at module level.

The word Object is illegal as a type name.

Declaring type members

A *member* is a variable declaration without the Dim, Private, Public, or Static keywords. A *member* cannot be declared to be Private, Public, or Static; it's automatically Public.

Each *member* statement declares one variable.

The data type of a *member* can be any of the scalar data types, a Variant, a fixed array, or any other user-defined data type. It cannot be the same data type as that being defined by the current Type statement.

A *member* declared as Variant can hold any scalar value, an array (fixed or dynamic), a list, or a reference to a user-defined object, a product object, or an OLE Automation object. The following rules apply to type instances that have Variant members containing arrays, lists, or objects:

- You cannot assign a type instance containing a dynamic array or a list to another type instance.
- You cannot use the Put statement to write data to a file from a type instance containing a dynamic array, a list, or an object.
- When you assign a type instance containing an object to another type instance, LotusScript increments the internal reference count of the object.

A *member* can use any LotusScript keyword, except Rem, as its name.

Declaring a type variable
A user-defined data type name is used in variable declarations in the same way as any other data type. The common variable declaration has the syntax:

Dim *varName* **As** *typeName*

This declaration declares a variable of the type *typeName* and initializes the members of the new variable. The initial values of the members are the same as for ordinary variables:

- Numeric data types (Integer, Long, Single, Double, Currency): 0
- Variants: EMPTY
- Strings, fixed-length: A string filled with the Null character Chr(0).
- Strings, variable-length: The empty string (“”).

If a member is itself a user-defined data type, then it is assigned initial values in the same manner.

Referring to type members
Refer to members of a type using dot notation, in the form *varName.memberName*. Spaces, tabs, and newline characters are legal on both sides of the period (after *varName* and before *memberName*).

Member references can also include array subscripts if the member is an array.

Examples: Type statement

Example 1

```
' Define a type with members to hold name, area code,
' and 7-digit local phone number.
Type phoneRec
   name As String
   areaCode As Integer
   phone As String * 8
End Type

Dim x As phoneRec              ' x is a variable of type phoneRec.
x.name$ = "Rory"               ' Assign values to x's members.
x.areaCode% = 999
x.phone$ = "555-9320"

Print "Call " & x.name$ & " at " & Str$(x.areaCode%) & "-" & x.phone%
Output:
' Call Rory at 999-555-9320"
```

Example 2

```
' Create an array to hold five instances of phoneRec.
Dim multiX(5) As phoneRec
multiX(2).name$ = "Maria"       ' Assign values.
multiX(2).areaCode% = 212
multiX(2).phone$ = "693-5500"

 ' Retrieve data from a type member.
Dim phoneLocalHold As String * 8
phoneLocalHold$ = multiX(2).phone$
Print phoneLocalHold$
' Output:
' 693-5500
```

Example 3

```
' To maintain a file that contains a phone list,
' read all of the data from the file into LotusScript.
' The data fills a list in which each element
' is an instance of the defined type.

' Create a list to hold records from the file.
Dim phoneList List As phoneRec

' Declare a phoneRec variable to hold
' each record from the file in turn. Open the file.
Dim tempRec As phoneRec
Open "c:\phones.txt" For Random Access Read Write _
   As #1 Len = Len(tempRec)

' Read the file and store the records in the list.
Dim recNum As Integer
```

```
recNum% = 1
While EOF(1) = FALSE
   Get #1, recNum%, tempRec
   phoneList(tempRec.Name$) = tempRec
   recNum% = recNum% + 1
Wend
Close #1
' Note that the Get statement automatically fills each member of the
' tempRec variable. Since tempRec and the elements of phoneList are
' both of data type phoneRec, tempRec can be assigned to any element
' of phoneList without reference to its members, which LotusScript
' copies automatically.
```

TypeName function

Returns a string identifying the data type of the value of an expression.

Syntax

TypeName (*expr* **)**

Elements

expr

Any expression.

Return value

Value of expr	*Return value*	*Storage of variable*
EMPTY	"EMPTY"	In Variant only
NULL	"NULL"	In Variant only
Integer	"INTEGER"	
Long	"LONG"	
Single	"SINGLE"	
Double	"DOUBLE"	
Currency	"CURRENCY"	
Date	"DATE"	In Variant only
String	"STRING"	
NOTHING	"OBJECT"	
OLE object	"OBJECT"	In Variant only
OLE error	"ERROR"	In Variant only
Boolean	"BOOLEAN"	In Variant only

continued

Value of expr	*Return value*	*Storage of variable*
V_UNKNOWN (OLE value)	"UNKNOWN"	In Variant only
User-defined object or product object	The name of the object's class, as an uppercase string. For example, for an object of the Employee class, LotusScript returns "EMPLOYEE."	
List	The name of the list's data type, plus the word "LIST," all as an uppercase string. For example, for a list of type String, LotusScript returns "STRING LIST."	
Array	The name of the array's data type as an uppercase string, followed by parentheses enclosing one space. For example, for an integer array, LotusScript returns "INTEGER( )."	

Examples: TypeName function

```
Dim a As Variant
Print TypeName(a)     ' Prints "EMPTY"
a = 1
Print TypeName(a)     ' Prints "INTEGER"
a = "hello"
Print TypeName(a)     ' Prints "STRING"
Dim b As String
Print TypeName(b$)    ' Prints "STRING"

' Arrays
Dim arrayl(1 To 4) As Long
Print TypeName(arrayl&)  ' Prints "LONG( )"
Dim arrayV(1 To 4)
Print TypeName(arrayV)   ' Prints "VARIANT( )"
Dim y As Variant
y = arrayl
Print TypeName(y)        ' Prints "LONG( )"

' Lists
Dim listStr List As String
Print TypeName(listStr$) ' Prints "STRING LIST"
Dim listVar List
Print TypeName(listVar)  ' Prints "VARIANT LIST"
Dim p As Variant
p = listStr$
Print TypeName(p)        ' Prints "STRING LIST"

' Class instances
Class Employee
   ' ... class definition
End Class
```

```
Dim temp As Employee
Print TypeName(temp)     ' Prints "EMPLOYEE"
Set hire = New Employee
Print TypeName(hire)     ' Prints "EMPLOYEE"
Dim emps(3) As Employee
Print TypeName(emps())   ' Prints "EMPLOYEE( )"

' OLE class instances
Set cal = CreateObject("dispcalc.ccalc")
Print TypeName(cal)      ' Prints "OBJECT"
```

UBound function

Returns the upper bound for one dimension of an array.

Syntax

UBound (*arrayName* [**,** *dimension*] **)**

Elements

arrayName

The name of an array.

dimension

Optional. An integer argument that specifies the array dimension for which you want to retrieve the upper bound.

Return value

UBound returns an Integer.

Usage

The default value for *dimension* is 1.

LotusScript sets the upper bound for each array dimension when you declare a fixed array, or when you use ReDim to define the array dimensions of a dynamic array.

Examples: UBound function

```
Dim maxima(10 To 20)
Dim upperBound As Integer
upperBound% = UBound(maxima)
Print upperBound%
' Output:
' 20
```

UCase function

Converts all alphabetic characters in a string to uppercase, and returns the resulting string.

Syntax

UCase[**$**] **(** *expr* **)**

Elements

expr

For UCase, any numeric or string expression. For UCase$, any Variant or string expression.

Return value

UCase returns a Variant of DataType 8 (String). UCase$ returns a String.

UCase(NULL) returns NULL. UCase$(NULL) returns an error.

Usage

The function has no effect on non-alphabetic characters.

Examples: UCase function

```
' Convert a string to uppercase.
Dim upperCase As String
upperCase$ = UCase$("abc") ' Assign the value "ABC"
```

UChr function

Returns the character represented by a Unicode numeric character code.

Syntax

UChr[**$**] **(** *longExpr* **)**

Elements

longExpr

Any expression with a numeric value between 0 and 65535, inclusive.

Return value

UChr and UChr$ return the Unicode character corresponding to the value of *longExpr*.

UChr returns a Variant of DataType 8 (String). UChr$ returns a String.

Examples: UChr function

```
Dim azAlphabet As String
Dim letterCode As Long

' Iterate through the Unicode values for a through z,
' appending each corresponding letter to azAlphabet.
For letterCode& = Uni("a") To Uni("z")
   azAlphabet$ = azAlphabet$ + UChr$(letterCode&)
Next
Print azAlphabet$     ' Prints abcdefghijklmnopqrstuvwxyz
```

Uni function

Returns the Unicode numeric character code for the first character in a string.

Syntax

Uni (*stringExpr* **)**

Elements

stringExpr
Any string expression.

Return value

Uni returns a Long.

Usage

If *stringExpr* is NULL or the empty string (""), the function returns an error.

Examples: Uni function

```
' Print the Unicode character codes for A and a.
Dim x As Long, y As Long
x& = Uni("A")
y& = Uni("a")
Print x&; y&     ' Prints 65  97
```

Unlock statement

See Lock and Unlock Statements.

Use statement

Loads a module containing Public definitions needed by the module being compiled.

Syntax

Use *useScript*

Elements

useScript

A String literal, or a constant containing a String value, specifying the module to load.

The Lotus product that you're using determines whether *useScript* must be compiled before use. Consult the product documentation for more information.

Usage

The Use statement can appear only at module level, before all implicit declarations within the module.

Loading a used module

Whenever LotusScript loads a module that contains a Use statement, LotusScript executes the Use statement before initializing the module and executing the module's Initialize sub, if the module contains one.

Referring to Public names in a used module

A used module remains loaded until it is explicitly unloaded. When a module is unloaded, references to Public names defined in that module become invalid and result in run-time errors.

Declaring Public names

A module's Public names are not visible to other modules until the first module is used. Multiple Public definitions for the same name cannot be loaded at the same time.

Using modules is transitive: if module A uses module B, and B uses C, then the Public names in C are visible in A.

Use statements must not contain circular references at compile time. If A uses B, then B or any module that B uses by transitivity cannot use A.

Examples: Use statement

```
Use "PreModule"
' The previously defined module PreModule is loaded.
' Any Public definitions in PreModule are available in
' the module where the Use statement appears.
```

UseLSX statement

Loads a LotusScript extensions (lsx) file containing Public definitions needed by the module being compiled.

Syntax

UseLSX *lsxLibraryName*

lsxLibraryName

A string literal specifying the lsx file to load, either a name prepended with an asterisk or the full path name of the file. If you specify a name prepended with an asterisk (for example, "*LSXODBC"), the file is determined by searching the registry, initialization file, or preferences file, depending on the client platform. The Windows 95 registry, for example, might contain an entry for HKEY_LOCAL_MACHINE, SOFTWARE, Lotus, Components, LotusScriptExtensions, 2.0, LSXODBC, whose value is "c:\notes95\nlsxodbc.dll."

Usage

LotusScript registers the Public classes defined in the lsx file for use in the module containing the UseLSX statement. Other modules that use this containing module can also access these Public classes.

Examples: UseLSX statement

```
UseLSX "appdll"
' The file appdll is loaded. Public definitions in the file
' are available to the module where the UseLSX statement appears.
```

UString function

Returns a string of identical characters. You can specify the repeating character either by its Unicode numeric code, or as the first character in a string argument.

Syntax

UString[**$**] **(** *stringLen* **,** { *charCode* | *stringExpr* } **)**

Elements

stringLen

A numeric expression whose value is the number of characters to put in the returned string. LotusScript rounds *stringLen* to the nearest integer.

charCode

A numeric expression whose value specifies the Unicode numeric character code for the repeating character. LotusScript rounds *charCode* to the nearest integer.

Unicode codes range from 0 through 65535 inclusive. The Uni function returns the Unicode code for a given character.

stringExpr

Any string expression. The first character in this string is the character to be used for the repeating character.

Return value

UString returns a Variant of DataType 8 (String). UString$ returns a String.

Usage

If the value of *charCode* is less than 0 or greater than 65535, the function returns an error.

Examples: UString function

```
Dim stars As String, moreStars As String
stars$ = UString$(4, Uni("*"))
moreStars$ = UString$(8, "*chars")
Print stars$, moreStars$   ' Prints ****      ********
```

Val function

Returns the numeric value represented by a string.

Syntax

Val (*stringExpr* **)**

Elements

stringExpr

Any string expression that LotusScript can interpret as a numeric value. It can contain any of the following kinds of characters.

- Digits (0 1 2 3 4 5 6 7 8 9)
- Other characters in hexadecimal integers (a b c d e f A B C D E F)
- Sign characters (+ -)
- Decimal point (.)
- Exponent characters (E e D d)
- Prefix characters in binary, octal, and hexadecimal integers (& B O H)

Return value

Val returns the converted part of *stringExpr* as a Double.

Usage

Val strips out spaces, tabs, carriage returns, and newlines from *stringExpr*. It starts converting from the beginning of the string and stops when it encounters a character other than those listed for *stringExpr* in the preceding list.

Examples: Val function

```
Dim hexVal As Double, streetNum As Double
' Assign the hexadecimal value FF (decimal 255).
hexVal# = Val("&HFF")
' Assign the value 106.
streetNum# = Val("     106 Main St.")
Print hexVal#; streetNum#
' Output:
' 255  106
```

Variant data type

Specifies a 16-byte variable that can contain data of any scalar type, an array, a list, or an object.

Usage

A variable that is declared without a data type or a suffix character is of type Variant.

Variant values are initialized to EMPTY.

A Variant variable can contain values of any scalar data type, or any of the following special values.

- Array: A declared array may be assigned to a Variant variable. The reverse is not true; for example, a Variant variable containing an array may not be assigned to a declared array variable.
- List: A list may be assigned to a Variant variable. The reverse is not true; for example, a Variant variable containing a list may not be assigned to a declared list variable.
- Object reference: A reference to any instance of a user-defined class or product class, or to an OLE Automation object, may be assigned to a Variant variable.
- Date/time value: An 8-byte floating-point value representing a date/time may be assigned to a Variant variable. The integer part represents a serial day counted from Jan 1, 100 AD. Valid dates are represented by integer numbers in the range -657434 (representing Jan 1, 100 AD) to 2958465 (representing Dec 31, 9999 AD). The fractional part represents the time as a fraction of a day, measured from time 00:00:00 (midnight on the previous day). In this representation of date/time values, day 1 is the date December 31, 1899.
- NULL: A Variant can take the value NULL either by explicit assignment, or by the evaluation of an expression containing NULL as an operand. (For most expressions, if one or both operands are NULL, the expression evaluates to NULL.)

- EMPTY: In expressions, EMPTY is converted to 0 for numeric operations, and to the empty string ("") for string operations. Variants take the value EMPTY only upon initialization, or upon assignment from another Variant whose value is EMPTY.

A Variant cannot contain an instance of a user-defined type.

To determine the data type of the value in a Variant variable, use the DataType or TypeName function.

LotusScript aligns Variant data on an 8-byte boundary. In user-defined data types, declaring variables in order from highest to lowest alignment boundaries makes the most efficient use of data storage space.

Examples: Variant data type

```
' Explicitly declare a Variant variable.
Dim someV As Variant

' Use the Variant variable to hold a Currency value.
Dim price As Currency
price@ = 20.00
someV = price@
Print DataType(someV)   ' Prints 6 (Currency)

' Use the Variant variable to hold an object reference.
Class Product
   Sub Sell(toCustomer)
      ' ...
   End Sub
End Class
Dim knife As New Product
Set someV = knife
Call someV.Sell("Joe Smith")   ' Calls Product method

' Use the Variant variable to hold an array.
Dim salesArray()
ReDim salesArray(3)
salesArray(1) = 200
salesArray(2) = 350
salesArray(3) = 10
someV = salesArray
Print someV(1)       ' Prints 200

' Use the Variant variable to hold a list.
Dim customerList List
customerList("one") = "Butcher"
customerList("two") = "Baker"
someV = customerList
Print someV("one")   ' Prints Butcher
```

Weekday function

Returns the day of the week, an integer from 1 to 7, for a date/time argument.

Syntax

Weekday (*dateExpr* **)**

Elements

dateExpr

Any of the following kinds of expression:

- A valid date/time string of String or Variant data type. In a date/time string, LotusScript interprets a 2-digit designation of a year as that year in the twentieth century. For example, 17 and 1917 are equivalent year designations.
- A numeric expression whose value is a Variant of DataType 7 (Date/Time).
- A number within the valid date range: -657434, representing Jan 1, 100 AD, to 2958465, representing Dec 31, 9999 AD.
- NULL.

Return value

Weekday returns an integer between 1 and 7.

The data type of the return value is a Variant of DataType 2 (Integer).

Weekday(NULL) returns NULL.

Usage

Sunday is day 1 of the week.

Examples: Weekday function

```
Dim x As Variant, wd As Integer
x = DateNumber(1993, 7, 7)
wd% = Weekday(x)
Print wd%
' Output:
' 4
```

While statement

Executes a block of statements repeatedly while a given condition is true.

Syntax

While *condition*

[*statements*]

Wend

Elements

condition

Any numeric expression. LotusScript interprets a value of 0 as FALSE, and interprets any other value as TRUE.

Usage

LotusScript tests *condition* before entering the loop and before each subsequent repetition. The loop repeats while *condition* is TRUE. When *condition* is FALSE, execution continues with the first statement following the Wend statement.

Examples: While statement

```
' While a user-specified interval (in seconds) is elapsing,
' beep and count the beeps. Then tell the user the number of beeps.

Dim howLong As Single, howManyBeeps As Integer
Function HowManyTimes (howLong As Single) As Integer
   Dim start As Single, finish As Single, counter As Integer
   start! = Timer
   finish! = start! + howLong!
   While Timer < finish!
      Beep
      counter% = counter% + 1
   Wend
   HowManyTimes = counter%
End Function
howLong! = CSng(InputBox _
   ("For your own sake, enter a small number."))
howManyBeeps% = howManyTimes(HowLong!)
MessageBox "Number of beeps:" & Str(howManyBeeps%)
```

Width # statement

Assigns an output width to a sequential text file.

Syntax

Width #*fileNumber* **,** *width*

Elements

#fileNumber

The file number that LotusScript assigned to the file when it was opened. The file must be open. You must include both the pound sign (#) and the file number.

width

An integer expression in the range 0 to 255, inclusive, that designates the number of characters LotusScript writes to a line before starting a new line. A *width* of 0, the default, specifies an unlimited line length.

Usage

If data to be written would cause the width of the current line to exceed the Width # setting, that data is written instead at the beginning of the next line.

The Print # statement is the only output statement affected by the Width # statement. Write # ignores the width set by Width #.

Examples: Width # statement

```
Dim fileNum As Integer
Dim fileName As String
fileName$ = "data.txt"
fileNum% = FreeFile()

Open fileName$ For Output As fileNum%
Width #fileNum%, 20

Print #fileNum%, "First line";

' The next data item, a long string, would extend the current line
' beyond 20 characters; so it is written to the next line in the file.
' An individual data item can not be split across lines;
' so the entire 33-character string is written to one line.
Print #fileNum%, "This will go on one line, though.";

' The next data item is written to the next line in the file
' because the current line is already wider than 20 characters.
Print #fileNum%, "But this is on another.";
Print #fileNum%, "The End";
Close fileNum%

' Output:
' First line
' This will go on one line, though.
' But this is on another.
' The End
```

With statement

Provides a shorthand notation for referring to members of an object.

Syntax

With *objectRef*

[*statements*]

End With

Elements

objectRef

An expression whose value is a reference to a user-defined object, a product object, or an OLE object.

Usage

The With statement lets you refer to the members of an object using a dot to represent the object name.

You can also use a dot outside of a With statement to represent the currently selected product object.

You cannot use a dot to refer to the selected product object in a With statement. LotusScript assumes that any member preceded by a dot is a member of *objectRef*.

You can nest With statements up to 16 levels.

LotusScript does not support entering a With statement using GoTo.

Reassigning the *objectRef* variable inside the With statement does not change the object referred to by the dot. However, any other operation reassigns the object. See the following example.

Examples: With statement

```
Class Employee
   Public empName As String
   Public status As Integer
   Sub SetName
      empName$ = InputBox$("Enter name:")
   End Sub
End Class

Dim emp As New Employee
Dim emp2 As New Employee
```

```
With emp
   Call .SetName    ' Calls InputBox$ to prompt for an employee
                   ' name to assign to emp.empName
   Set emp = emp2   ' Reassigns the emp object variable, to refer to
                    ' a different object (the same object that emp2
                    ' refers to)
   .status% = 1     ' Sets status of the object that emp referred to
                     ' when the With statement was entered.
   emp.status% = 0   ' Sets both emp.status and emp2.status, because
                     ' of the preceding Set statement
   Print .status% ; emp.status% ; emp2.status%
' Output:
' 1  0  0
End With
```

Write # statement

Writes data to a sequential text file with delimiting characters.

Syntax

Write #*fileNumber* [**,** *exprList*]

Elements

#fileNumber

The file number that LotusScript assigned to the file when it was opened. You must include both the pound sign (#) and the file number.

exprList

Optional. The list of String or numeric expressions to be written to the file, separated with commas.

If you omit *exprList*, Write # writes a blank line to the file.

The *exprList* can't include arrays, lists, type variables, or objects. The *exprList* can include individual array elements, list elements, or type members.

Usage

Use Write # only with files opened for Output or Append.

Use the Input # statement to read data written by Write #.

Write # ignores the file width set by the Width # statement. Data items are separated with commas, and a newline character is inserted after all data has been written to the file.

LotusScript inserts a "\n" character in any multiline string (for example, a string that you type in using vertical bars or braces). If you use the Print # statement to print the string to a sequential file, the \n is interpreted as a newline on all platforms. If you use Write # to write the string to a sequential file, the \n may not be interpreted as a newline on all platforms. Therefore, when reading a multiline string from a sequential file written by the Write # statement, use Input, not Line Input.

The following table shows how the Write # statement behaves with various data types specified in *exprList*.

Data type	*Write # statement behavior*
Numeric	Omits leading and trailing spaces.
String	Encloses all strings in double quotation marks. Pads fixed-length strings with spaces as needed.
Variant of DataType 7 (Date/Time)	Uses one of the following date formats: #yyyy-mm-dd hh:mm:ss# #yyyy-mm-dd# #hh:mm:ss# If either the date part or the time part is missing from the value, LotusScript writes only the part provided to the file.
Variant with the value EMPTY	Writes a comma without data to the file. If that variable is the last item on the line, the comma is omitted.
Variant with the value NULL	Writes the string NULL to the file.

Examples: Write # statement

```
Dim fileNum As Integer, empNumber As Integer, I As Integer
Dim fileName As String, empName As String
Dim empLocation As Variant
Dim empSalary As Currency

fileNum% = FreeFile()
fileName$ = "data.txt"

' Write out some employee data.

Open fileName$ For Output As fileNum%
Write #fileNum%, "Joe Smith", 123, "1 Rogers Street", 25000.99
Write #fileNum%, "Jane Doe", 456, "Two Cambridge Center", 98525.66
Write #fileNum%, "Jack Jones", 789, "Fourth Floor", 0
Close fileNum%

' Read it all back and print it.
Open fileName$ For Input As fileNum%

For I% = 1 To 3
   Input #fileNum%, empName$, empNumber%, empLocation, empSalary@
   Print empName$, empNumber%, empLocation, empSalary@
Next I%
```

```
Close fileNum%

' Output:
' LotusScript prints out the contents of the file C:\data.txt
' in groups of four values each. Each group consists of a String,
' an Integer, a Variant, and a Currency value, in that order.
```

Year function

Returns the year, as a 4-digit integer, for a date/time argument.

Syntax

Year (*dateExpr* **)**

Elements

dateExpr

Any of the following kinds of expressions:

- A valid date/time string of String or Variant data type. In a date/time string, LotusScript interprets a 2-digit designation of a year as that year in the twentieth century. For example, 17 and 1917 are equivalent year designations.
- A numeric expression whose value is a Variant of DataType 7 (Date/Time).
- A number within the valid date range: -657434, representing Jan 1, 100 AD, to 2958465, representing Dec 31, 9999 AD.
- NULL.

Return value

Year returns an integer between 100 and 9999.

The data type of the return value is a Variant of DataType 2 (Integer).

Year(NULL) returns NULL.

Examples: Year function

```
Dim x As Variant
Dim yy As Integer
x = DateNumber(1995, 4, 1)
yy% = Year(x)
Print yy%
' Output:
' 1995
```

Yield function and statement

Transfers control to the operating system during script execution.

Syntax
Yield

DoEvents is acceptable in place of Yield.

Return value
The Yield function returns 0 as an Integer value.

Usage
The Yield function and statement transfer control to the operating system, so that it can process the events in its queue. In Windows, the operating system does not return control until it has processed all outstanding events, including those generated by a SendKeys statement.

The Yield function and statement are legal within a procedure or a class. They are not legal at the module level.

You can call the function as either Yield or Yield().

Examples: Yield function and statement
Yield control to allow the user to perform one or more calculations. When the user is done, continue with the script.

The DoCalc sub uses a Shell statement to start the Windows calculator. The Shell statement returns the calculator task ID (also known as the module handle). In a While loop, the sub calls the GetModuleUsage Windows 3.1 API function, which returns the module reference count (how many instances of the calculator are currently running). The Yield statement yields control to the calculator. When the user closes the calculator, GetModuleUsage returns a reference count of 0, the While loop ends, and the sub displays an appropriate message.

If you remove the While loop (try it), the message box appears as soon as the calculator begins running. In other words, the script continues to execute without yielding control to the calculator.

```
' Declare the Windows 3.1 API function at the module level.
Declare Function GetModuleUsage Lib "Kernel" _
   (ByVal taskID As Integer) As Integer

Sub DoCalc
   Dim taskID As Integer
   ' Start the Windows calculator, returning its task ID.
   taskID% = Shell("calc.exe", 1)
   ' As long as the module is still running, yield.
   Do While GetModuleUsage(taskID%) > 0
      Yield
   Loop
   ' When the user closes the calculator, continue.
   MessageBox "Calculations done"
End Sub

DoCalc                ' Call the DoCalc sub.
```

Part 3
Appendixes

Appendix A
Language and Script Limits

This appendix describes LotusScript language limits of several kinds: for example, the legal ranges in data representation, the limits on numerical specifications within statements, and the maximum number of different kinds of elements that can be defined in a script.

Limits on numeric data representation

The following table lists the legal range of values for the numeric data types.

Data type	*Range*
Integer	-32,768 to 32,767
Long	-2,147,483,648 to 2,147,483,647
Single	-3.402823E+38 to 3.402823E+38 *Smallest non-zero value (unsigned):* 1.175494351E-38
Double	-1.7976931348623158E+308 to 1.7976931348623158E+308 *On UNIX platforms:* -1.797693134862315E+308 to 1.797693134862315E+308 *Smallest non-zero value (unsigned):* 2.2250738585072014E-308
Currency	-922,337,203,685,477.5808 to 922,337,203,685,477.5807 *On UNIX platforms:* -922,337,203,685,477.5666 to 922,337,203,685,477.5666 *Smallest non-zero value (unsigned):* .0001

The legal range of values of binary, octal, or hexadecimal integers is the range for Long integers (see the preceding table). The following table lists the maximum number of characters needed to represent integers in binary, octal, and hexadecimal notation. This is also the maximum number of characters that the Bin, Oct, or Hex function returns.

Integer type	*Maximum number of characters needed to represent a value*
Binary	32
Octal	11
Hexadecimal	8

Limits on string data representation

The following table lists the limits on representation of string data.

Item	*Maximum*
Number of strings	Limited by available memory
Total string storage	In a module, 32K characters (64K bytes). For strings generated during execution, storage is limited by available memory.
Length of a string literal	16,000 characters (32,000 bytes)
Length of a string value	32,000 characters (64,000 bytes)

Limits on array variables

The following table lists limits on representation of data by array variables.

Item	*Maximum or range*
Array storage size	64K bytes
Number of dimensions	8
Bounds of a dimension	-32,768 to 32,767 (the range of values of the Integer data type)
Number of elements	Determined by memory available for data, and by the storage size of each element of the array, which varies with the array data type. For example, a Long one-dimensional fixed array declared in module scope can have 16,128 elements. (The total storage size available for fixed-size data in module scope is 64K bytes, and a Long element requires 4 bytes for storage.)

Limits on file operations

The following table lists limits on miscellaneous items related to file operations and I/O.

Item	*Maximum*
Number of files open simultaneously	Determined by the product from which you start LotusScript
fileNumber in Open statement	255
recLen in Open statement	255
Line length of a line written by Write statement	255 characters
Number of items in Print, Write, or Input statement	255
Number of characters in *path* in MkDir, RmDir, or ChDir statement	128. This includes the drive specifier, if any.

Limits in miscellaneous source language statements

The following table lists limits on miscellaneous language elements.

Item	*Maximum*
Number of characters in a LotusScript identifier, not including a data type suffix character	40
Number of arguments in definition of a function or sub	31
Number of labels in an On...GoTo statement	255

Limits on compiler and compiled program structure

The following table lists limits on miscellaneous items related to compiling a script.

Item	*Maximum*
Number of lines per script, not including the contents of %Include files.	64K
Depth of nested %Include directives.	16
Number of compilation errors before the LotusScript compiler halts.	20
Number of symbols in a module's symbol table. (See "Number of symbols," below.)	Varies

continued

Item	*Maximum*
Number of recursive calls (recursion level for a given function).	Limited by available memory
Storage size of all data in a given scope. (See “Storage size of data,” below.)	Module: 64K bytes Class: 64K bytes Procedure: 32K bytes
Size of executable module code.	64K bytes

Number of symbols

A module symbol table can occupy up to 64K bytes. Each symbol occupies at least 10 bytes. Variables require the minimum space; about 6000 variables can be declared in a module if no functions or subs are declared. Function and sub declarations need more space; a maximum of about 450 functions or subs can be declared if no variables are declared. A module commonly includes symbols of both kinds, and the maximum number of each is interdependent. In any case it is large.

Storage size of data

The limits on the storage size of data in a given scope apply to fixed-size variables: scalar variables except for variable-length strings; user-defined type variables; and fixed arrays of these scalar variables and user-defined type variables. Depending on the order of declaration, alignment of variables on storage boundaries can take extra space. For example, an Integer variable is aligned on a 2-byte boundary, and a Long variable is aligned on a 4-byte boundary.

The maximum size of data in each dynamic variable (each variable-length string, each list, each dynamic array, and each instance of a class) is 64K. However, each such variable will use 4 bytes for data in the scope where it is declared.

Appendix B
Platform Differences

The LotusScript language and functionality on the OS/2 platform, the UNIX platform, and the Macintosh platform differ in various ways from the language and functionality described in the rest of this language reference. This appendix describes the differences.

OS/2 platform differences

Language construct differences

Command	Command-line arguments are not normally used on OS/2. However, if the Lotus product permits arguments, they are returned.
CreateObject	Not supported. Generates a run-time error.
GetObject	Not supported. Generates a run-time error.
Shell	The window style option is not supported for an OS/2 system application or for a user application that saves its environments via Profile. The default window style is normal with focus. Shell always returns a valid value greater than 31.

File system differences

LotusScript supports both HPFS and FAT file systems:

- The FAT file system supports conventional file names only. Conventional file names consist of up to 8 characters, a period separator, and up to 3 characters.
- The HPFS file system recognizes both conventional and long file names. Long file names can be up to 254 characters in length, including any number of periods. Blanks are supported if the file name is enclosed in double quotes. A file name consisting either of all periods or all blanks is not supported.

HPFS requires 500K of system memory. Each OS/2 PC must have at least 6MB of memory as a minimum requirement; otherwise performance will be adversely affected.

Files with long file names or blank spaces can be copied only to a diskette or disk formatted with FAT using the direct-manipulation method. Long file names are truncated to conventional file length when moved from a HPFS to a FAT file system.

The long file name is saved as an extended attribute until the file is copied back to an HPFS disk using the direct-manipulation method and the workplace shell. The use of HPFS files incorrectly transferred to a FAT file system results in a run-time error.

An asterisk (*) as a wildcard in a file name indicates that any character can occupy that position and all remaining character positions. A question mark (?) as a wildcard in a file name indicates that any character can occupy that position only.

File names are not case sensitive.

Other differences

OLE functions are not supported. This limitation affects CreateObject and GetObject.

OS/2 users can invoke Rexx applications from LotusScript.

UNIX platform differences

Language construct differences

ActivateApp	Not supported. Generates a run-time error.
ChDir	A run-time error is generated if LotusScript cannot interpret the argument to ChDir, for example if a drive letter is contained in the argument.
ChDrive	Generates a run-time error unless the drive argument is the empty string (“”), signifying the default drive.
CreateObject	Not supported. Generates a run-time error.
CurDir, CurDir$	Generates a run-time error unless the drive argument is the empty string (“”), signifying the default drive.
CurDrive, CurDrive$	Return the empty string (“”), since there are no drive letters on UNIX.
Date, Date$	For reasons of security and system integrity, only the superuser can change the date on a UNIX system. Attempting to change the date under any other username will generate a run-time error. Attempting to change the date while logged in as superuser will change the date system-wide.
Declare	The Pascal calling convention for external function calls is not supported. All external function calls must use the CDECL calling convention. Specifying an ordinal number (using the Alias clause) is not supported. This will return a run-time error at the point of the call to the illegally declared function.

continued

Dir, Dir$	Ignores the optional *attributeMask* argument. These functions behave as if all files have the attribute Normal. **Returns all files for "*.*", not just those containing ".". Returns only those files ending with a period for "*.", not every file without an extension.**
FileLen, Len, LenB, LenBP, LOF	Strings containing line terminators are smaller than on DOS/Windows platforms. The line terminator is one character (linefeed), not two. Therefore the return value of these functions will be smaller for strings on UNIX than on Windows.
GetFileAttr	Generates a run-time error if a drive letter is included in the argument. Does not return the following attributes: ATTR_HIDDEN, ATTR_ARCHIVE, ATTR_VOLUME, ATTR_SYSTEM.
GetObject	Not supported. Generates a run-time error.
Input #, Input, Input$, InputB, InputB$, Line Input, Print, Write #	Compiled scripts using these constructs may be platform-specific, since file data is stored in a platform-specific manner. UNIX character set, byte order, line terminator, and numeric precision specifics may affect the portability of scripts using these functions.
IsObject, IsUnknown	See "Other differences," below.
Open, Lock, Unlock	No explicit or implicit file locking is supported on UNIX. This implies the following: LotusScript for UNIX allows the user to copy, open, etc., a file that is already opened for reading. Thus, the Name statement works differently on UNIX. The Open statement may specify only Shared as its lock status. Lock Read, Lock Write, and Lock Read Write will cause a run-time error. The Lock and Unlock statements will cause a run-time error.
SendKeys	Not supported. Generates a run-time error.
SetFileAttr	Ignores the attributes ATTR_HIDDEN, ATTR_ARCHIVE, and ATTR_VOLUME.
Shell	Window styles are ignored.
Time, Time$	For reasons of security and system integrity, only a superuser can change the time on a UNIX system. Attempting to change the time under any other username will generate a run-time error. Attempting to change the time while logged in as superuser will change the time system-wide.

File system differences

LotusScript respects all aspects of UNIX file system security. This difference affects Kill, Open, and RmDir.

There are no drive letters on UNIX. All devices reside under the root directory. If you use a pathname containing a drive letter, LotusScript may return an error. For the %Include directive, this is a compiler error; for all other uses, this is a run-time error. (Note that since UNIX allows ":" in file names, the statement Dir$("a:") is legal. It searches the current directory for a file named a:.)

UNIX uses the "/" character (slash) as the directory separator while DOS/Windows platforms use "\" (backslash). LotusScript supports the use of slash and backslash, with the following restrictions:

- String literals. If a slash is used in a string literal that is a pathname argument, the .LSO file generated will not run on other platforms, unless that platform supports slash (for example, the UNIX platform).
- String variables. If you assign a string literal containing a slash to a variable, and then pass the variable as a pathname argument, a run-time error occurs if the platform does not support slash pathnames (for example, the DOS/Windows platform).

UNIX allows a wider variety of characters in pathnames than DOS/Windows platforms. For example, more than one "." may appear in a valid UNIX pathname.

LotusScript cannot use UNIX filenames (as opposed to pathnames) that contain the "\" character, since this character is always a path separator on other platforms.

UNIX uses the linefeed (ASCII 10) character as the line terminator. Other platforms use other characters. This difference means that files manipulated with the same LotusScript code, but executed on different platforms, may have different sizes. For instance, the MacIntosh platform uses the carriage return character as the line terminator, so text files written on that platform have the same length as files written on UNIX. Since the Windows platform uses a two-character sequence, text files written there are larger than text files written on UNIX, given identical source code.

Other differences

Function aliasing with ordinal numbers (using the Alias clause in the Declare statement) is not possible on UNIX, because UNIX has no notion of numbering the routines in a shared library.

Where wildcards are permitted in file path strings, LotusScript supports the use of UNIX regular expressions in addition to the "*" and "?" characters. However, using regular expressions in file path strings makes the script platform-dependent.

The Like operator does not use use the same regular expression syntax as the UNIX shell. It uses LotusScript regular expressions.

OLE is not supported on LotusScript Release 3.0 for UNIX platforms. This difference affects CreateObject, GetObject, IsObject, and IsUnknown. The CreateObject and GetObject functions will raise run-time errors when executed on UNIX platforms. The IsObject function tells if a variable refers to a native or product object, but not an OLE object, since OLE objects don't exist on the UNIX platform. The IsUnknown function always returns FALSE on UNIX, since there is no way for a Variant expression to receive the V_UNKNOWN value.

Macintosh platform differences

Language construct differences

ChDir	Macintosh hard drive specifications are supported; for example, "Hard drive:folder1: folder2:". DOS drive specifications, such as "C:\", are not suppported.
ChDrive	Generates a run-time error unless the drive argument is the empty string (""), signifying the default drive. To change the drive, use ChDir.
Command	Command line arguments are not normally used on the Macintosh. However, if the Lotus product permits arguments, they are returned.
CurDir	Generates a run-time error unless the drive argument is defaulted or explicitly specified as the empty string (""), signifying the default drive.
CurDrive	Returns the empty string (""), since there are no drive letters on the Macintosh.
Declare	The Pascal calling convention for external function calls is not supported.
Dir	Ignores the attributes Hidden Files, Volume Label, and System. Does not return the directory specifications "." and "..". Returns all files for "*.*", not just those containing ".". Returns only those files ending with a period for "*.", not every file without an extension.
Environ	Returns an empty string. Generates a run-time error only if an illegal argument is passed, such as a variable number greater than the legal limit.
FileLen	Files containing line terminators are smaller than on DOS platforms, because the line terminator is one character, not two.
GetFileAttr	Does not return the following attributes: ATTR_ARCHIVE, ATTR_VOLUME, ATTR_SYSTEM
Len, LenB	Strings that have been read from files containing line terminators are smaller than on DOS platforms, because the line terminator is one character, not two.
Lock	Open files can be manipulated (copied, opened, etc.).
Open	Open files can be manipulated (copied, opened, etc.).
SendKeys	Not supported. Generates a run-time error.
SetFileAttr	Generates a Permission Denied error if passed the attribute ATTR_ARCHIVE or ATTR_SYSTEM.
Unlock	Open files can be manipulated (copied, opened, etc.).

File system differences

Macintosh-style pathnames are assumed unless the pathname contains a backslash. If the pathname contains a backslash, then a DOS-style pathname is assumed.

There are no drive letters on the Macintosh. All devices reside under the root directory. If you use a pathname containing a drive letter, LotusScript may return an error. For the %Include directive, this is a compiler error; for all other uses, this is a run-time error.

Files are not limited to DOS naming rules (8-character name plus 3-character extension).

The Macintosh does not store a default directory for each drive. It maintains only one current directory, not one per drive as in DOS. Drive names can be up to 27 characters in length. This limitation affects ChDir, ChDrive, and CurDir.

The Macintosh does not recognize the directory specifications "." and "..". This limitation affects the Dir function.

The Macintosh does not use the file system attributes Volume, Archive, and System. This limitation affects Dir, GetFileAttr, and SetFileAttr.

Macintosh uses the carriage return (ASCII 13) character as the line terminator. Other platforms use other characters. This difference means that files and strings manipulated with the same LotusScript code but executed on different platforms may have different sizes. For instance, the UNIX platform uses a single character (linefeed) as the line terminator, so text files written on that platform have equal length to those written on Macintosh. Since the Windows platform uses a two-character sequence, text files written there are larger than text files written on Macintosh, given identical source code. This difference affects FileLen, Len, LenB, and LenBP.

Macintosh permits files that are open for reading to be manipulated (copied, opened, etc.) by another application. A file opened for output by LotusScript is locked; other applications cannot open or copy the file, but can move or rename it. Lock and Unlock work only on shared volumes; the file being locked must be on a server or file sharing must be turned on for a local volume ("Sharing Setup" on the control panel). This difference affects Open, Lock, and Unlock.

Other differences

Function aliasing with ordinal numbers (using the Alias clause in the Declare statement) is not possible on the Macintosh PC.

There are no system environment variables on the Macintosh. This limitation affects Environ.

Please be careful not to remove the floating page footer when you are replacing this text with your first paragraph.

Appendix C
LotusScript/Rexx Integration

If you use LotusScript in OS/2, you can use the LTSRXO10.DLL LSX to invoke applications written in the OS/2 Procedures Language 2/REXX (referred to as REXX in the rest of this documentation). You can execute a single REXX statement using REXXFunction, or execute an external REXX command file with REXXCmd.

REXXCmd function

Executes a REXX command file from within a LotusScript application, passing any needed arguments.

Syntax

REXXCmd (*commandFile* [**,** *parmList*] **)**

Elements

returnString

A string containing the value returned by the REXX command file.

commandFile

A string specifying the name of the REXX command file, optionally including a path specification.

parmList

Up to 20 parameters of any data type recognized by REXX, except arrays. Separate parameters with commas; enclose strings in quotation marks.

Usage

The REXXCmd function must be preceded with a UseLSX statement that indicates the location of the LotusScript/REXX LSX (LTSRXO10.DLL).

Examples: REXXCmd Function

```
' LotusScript application that uses REXXCmd to execute a REXX command
file.

' A sample REXX command file follows, showing how REXX accepts the

' parameters passed by LotusScript.

'

' Indicate the location of the LSX.
```

```
UseLSX "C:\LSX\LTSRXO10.DLL"

Function InvokeREXX
   Dim cmdFile As String
   Dim returnValue As Variant
   Dim parmOne As String, parmTwo As String

   parmOne = "Parameter 1"
   parmTwo = "Parameter 2"
   cmdFile = "RXSAMPLE.CMD"

' Invoke the REXX command file RXSAMPLE.CMD and store the return value
' in returnValue.
   returnValue = REXXCmd ( cmdFile, parmOne, parmTwo )
End Function

/*********************************************************************
***/
/*                          RXSAMPLE.CMD
  */
/*********************************************************************
***/
address CMD
/*********************************************************************
***/
/* Get the argument string passed by LotusScript.
 */
/*********************************************************************
***/
vari = ARG(1)
/*********************************************************************
***/
/* Now get the individual arguments; we know there are two.
 */
```

```
/*********************************************************************
***/

parse var vari argument1 ',' argument2

/*********************************************************************
***/

/* Continue processing. . .when done, return the string "OK."
  */

/*********************************************************************
***/

return "OK"

/*********************************************************************
***/

/* Exit the command file, returning control to LotusScript.
 */

/*********************************************************************
***/

exit
```

REXXFunction function and statement

Executes a single REXX statement from within a LotusScript application, optionally returning a value to LotusScript.

Function Syntax

REXXFunction ("RETURN *REXXStatement***")**

Statement Syntax

REXXFunction ("*REXXStatement***")**

Elements

returnString

A string containing the value returned by the REXX statement. Use this only when using REXXFunction as a function.

RETURN

Indicates that a value will be returned by the REXX statement. Use this only when using REXXFunction as a function; be sure to include it within the quotation marks enclosing *REXXStatement.*

REXXStatement

A string consisting of a valid REXX statement, enclosed in quotation marks.

Usage

The REXXFunction function and statement must be preceded with a UseLSX statement that indicates the location of the LotusScript / REXX LSX (LTSRXO10.DLL).

Examples: REXXFunction function and statement

```
' LotusScript application that uses REXXFunction to
' execute a REXX statement.
'
' Indicate the location of the LSX.
UseLSX "C:\LSX\LTSRXO10.DLL"
'
Function InvokeREXX
   Dim driveC As Variant
'
' Use REXXFunction as a statement to execute a REXX statement.
   REXXFunction _
   ( "Call RxFuncAdd 'SysLoadFuncs', 'RexxUtil', 'SysLoadFuncs' " )
   REXXFunction ( "Call SysLoadFuncs" )
'
' Now use REXXFunction as a function to execute another REXX statement
and
' return a value to LotusScript, storing it in the driveC variable.
   driveC = REXXFunction ( "RETURN SysDriveInfo ( ""C:"" ) " )
'
End Function
```

REXXLTS.EXE call

Invokes a LotusScript application**.**

Syntax

"Call REXXLTS.EXE" *scriptName functionName parmList*

Elements

scriptName

A string specifying the name of the LotusScript file being invoked. Include both the file name and extension; the extension must be either .LSO or .LSS. If you use the actual name of a script, include it within the quotation marks enclosing the call to REXXLTS.EXE; if you use a variable to represent *scriptName,* it should appear outside of the quotation marks.

functionName

A string specifying the name of the LotusScript function to execute; this function is defined within the specified LotusScript file. If you use the actual name of the script and function, include them within the quotation marks enclosing the call to REXXLTS.EXE; if you use variables to represent *scriptName* and *functionName,* they should appear outside of the quotation marks. If *scriptName* is represented with a variable, then *functionName* must also be represented with a variable.

parmList

Up to 20 parameters of data type Short, Long, Single, Double, or String. Separate parameters with spaces; enclose strings in quotation marks.

Usage

Use the REXX Call command to invoke LotusScript from a REXX command file. LotusScript return codes will be passed back to the command file; these return codes are described in the LotusScript LSERR.LSS file.

Examples: REXXLTS.EXE call

```
/*********************************************************************
***/

/* Sample REXX command file that uses REXXLTS.EXE to invoke a
  */

/* LotusScript file called TEST.LSS, passing two parameters to it.
 */

/*********************************************************************
***/

address CMD
```

```
/*****************************************************************
***/

/* Initialize the parameters.
 */

/*****************************************************************
***/

parm1 = "Parameter 1"

parm2 = "Parameter 2"

/*****************************************************************
***/

/* Invoke REXXLTS.EXE and call the LotusScript file.
 */

/* Since scriptName and functionName are represented by
  */

/* variables, do not include them within the quotation marks.
 */

/*****************************************************************
***/

"CALL REXXLTS.EXE TEST.LSS Function1" parm1 parm2

/*****************************************************************
***/

/* Exit the command file.
 */

/*****************************************************************
***/

exit
```

Index

Symbols

A

B

C

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y