Reliance
Industrial SCADA/HMI system

Reliance 4

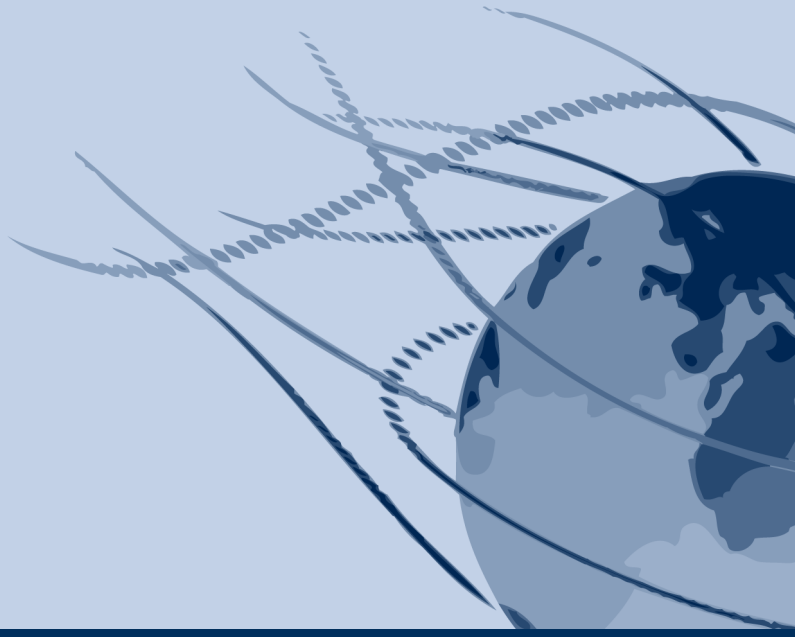
SCRIPTS

GEOVAP

Reliance
Industrial SCADA/HMI system

Reliance 4

SCRIPTS



GEOVAP

GEOVAP, spol. s r.o.
Cechovo nabrezi 1790
530 03 Pardubice
Czech Republic
+420 466 024 618
http://www.geovap.cz

Table of Contents

1 Introduction

About Visual Basic Script

Visual Basic Script, or *VBScript* for short, is a scripting language developed by *Microsoft*. The code written using a scripting language is interpreted when executed. This is in contrast to programming languages where the code must be compiled and linked before it can be executed. *VBScript* is designed for writing scripts for programs running on the Windows operating systems.

About this document

This document contains an introduction to *VBScript*, provides help on the basic procedures and functions and points out some basic syntax rules. For detailed information on *VBScript*, see the original help.

In addition to help on *VBScript*, this document also contains detailed information on Reliance-defined objects, which enable you to access the *Reliance* runtime environment from scripts.

Information on VBScript and Reliance is also available on the Internet at:

http://msdn2.microsoft.com/en-us/library/ms950396.aspx

www.reliance.cz

2 VBScript Language Reference

2.1 Syntax of procedure and function calls

A function is a routine that returns a value when it executes. A procedure is a routine that does not return a value. In *VBScript*, there is a syntax difference between calling a **procedure** and **function**.

When calling a procedure with parameters, the parameters cannot be enclosed in parentheses.

When calling a function with parameters, the parameters cannot be enclosed in parentheses if the return value is not processed. Otherwise, the parameters must be enclosed in parentheses.

For detailed information on procedures and functions in *VBScript*, see the original help.

Example

```
Dim ArrayOfNumbers(10)
Dim Response
' A procedure, the parameters are not enclosed in parentheses.
Erase ArrayOfNumbers
' The return value will be processed,
' the parameters are enclosed in parentheses.
Response = MsgBox("Continue?", vbYesNo, "Confirm")
If Response = vbYes Then
  ' ...
Else
  ' ...
End If
' The return value will not be processed,
' the parameters are not enclosed in parentheses.
MsgBox "Finished.", vbOKOnly, "Information"
```

2.2 Working with properties and methods of objects

VBScript enables you to work with objects. Using a reference to an object, it is possible to access properties and call methods (procedures and functions) of the object.

Syntax

```
MyObject.Function1
```

When accessing properties of an object, separate the object reference and the name of the property by a period. When calling a method of an object, separate the object reference and the name of the method by a period. Moreover, a method call must comply with the syntax of procedure and function calls.

For detailed information on objects in *VBScript*, see the original help.

Example

```
Dim fso, MyFile
' Create an object for working with files.
Set fso = CreateObject("Scripting.FileSystemObject")
' Create the text file C:\testfile.txt
' by calling a method, which returns another object.
Set MyFile = fso.CreateTextFile("C:\testfile.txt", True)
' Write a single line to the file by calling the WriteLine method.
MyFile.WriteLine("This is a test. ")
MyFile.Close ' Close the file by calling the Close method.
Set MyFile = Nothing
Set fso = Nothing
```

2.3 Data Type Functions

- **CBool** Function
- **CByte** Function
- **CCur** Function
- **CDate** Function
- **CDbl** Function
- **CInt** Function
- **CLng** Function
- **CSng** Function
- **CStr** Function
- **Fix** Function
- **Int** Function
- **IsArray** Function
- **IsDate** Function
- **IsEmpty** Function
- **IsNull** Function
- **IsNumeric** Function
- **IsObject** Function
- **TypeName** Function
- **VarType** Function

2.3.1 CBool Function

Returns an expression that has been converted to a **Variant** of subtype **Boolean**.

Syntax

CBool(*Expression***)**

The *Expression* argument is any valid expression.

Remarks

If *Expression* is zero, **False** is returned; otherwise, **True** is returned. If *Expression* can't be interpreted as a numeric value, a run-time error occurs.

Data Type Functions

Conversion Functions

Example

The following example uses the **CBool** function to convert an expression to a **Boolean**. If the expression evaluates to a nonzero value, **CBool** returns **True**; otherwise, it returns **False**.

```
Dim A, B, Check
A = 5
B = 5                 ' Initialize variables.
Check = CBool( A = B) ' Check contains True.
A = 0                 ' Define variable.
Check = CBool( A)     ' Check contains False.
```

2.3.2 CByte Function

Returns an expression that has been converted to a **Variant** of subtype **Byte**.

Syntax

CByte(*Expression***)**

The *Expression* argument is any valid expression.

Remarks

In general, you can document your code using the subtype conversion functions to show that the result of some operation should be expressed as a particular data type rather than the default data type. For example, use **CByte** to force byte arithmetic in cases where currency, single-precision, double-precision, or integer arithmetic normally would occur.

Use the **CByte** function to provide internationally aware conversions from any other data type to a **Byte** subtype. For example, different decimal separators are properly recognized depending on the locale setting of your system, as are different thousand separators.

If *Expression* lies outside the acceptable range for the **Byte** subtype, an error occurs.

Data Type Functions

Conversion Functions

Example

The following example uses the **CByte** function to convert an expression to a byte.

```
Dim MyDouble, MyByte
MyDouble = 125.5678      ' MyDouble is a Double.
MyByte = CByte( MyDouble) ' MyByte contains 126.
```

2.3.3 CCur Function

Returns an expression that has been converted to a **Variant** of subtype **Currency**.

Syntax

CCur(*Expression***)**

The *Expression* argument is any valid expression.

Remarks

In general, you can document your code using the subtype conversion functions to show that the result of some operation should be expressed as a particular data type rather than the default data type. For example, use **CCur** to force currency arithmetic in cases where integer arithmetic normally would occur.

You should use the **CCur** function to provide internationally aware conversions from any other data type to a **Currency** subtype. For example, different decimal separators and thousands separators are properly recognized depending on the locale setting of your system.

Data Type Functions

Conversion Functions

Example

The following example uses the **CCur** function to convert an expression to a Currency.

```
Dim MyDouble, MyCurr
MyDouble = 543.214588 '  MyDouble is a Double.
' Convert result of MyDouble * 2 (1086.429176) to a Currency (1086.4292).
MyCurr = CCur( MyDouble * 2)
```

2.3.4 CDate Function

Returns an expression that has been converted to a **Variant** of subtype **Date**.

Syntax

CDate(*Date***)**

The *Date* argument is any valid date expression.

Remarks

Use the IsDate function to determine if *Date* can be converted to a date or time. **CDate** recognizes date literals and time literals as well as some numbers that fall within the range of acceptable dates. When converting a number to a date, the whole number portion is converted to a date. Any fractional part of the number is converted to a time of day, starting at midnight.

CDate recognizes date formats according to the locale setting of your system. The correct order of day, month, and year may not be determined if it is provided in a format other than one of the recognized date settings. In addition, a long date format is not recognized if it also contains the day-of-the-week string.

Data Type Functions

Conversion Functions

Example

The following example uses the **CDate** function to convert a string to a date. In general, hard coding dates and times as strings (as shown in this example) is not recommended. Use date and time literals (such as `#10/19/1962#, #4:45:23 PM#`) instead.

```
Dim MyShortTime, MyDate, MyTime
MyDate = "October 19, 1962" ' Define date.
MyShortDate = CDate(MyDate) ' Convert to Date data type.
MyTime = "4:35:47 PM"       ' Define time.
MyShortTime = CDate(MyTime) ' Convert to Date data type.
```

2.3.5 CDbl Function

Returns an expression that has been converted to a **Variant** of subtype **Double**.

Syntax

CDbl(*Expression***)**

The *Expression* argument is any valid expression.

Remarks

In general, you can document your code using the subtype conversion functions to show that the result of some operation should be expressed as a particular data type rather than the default data type. For example, use **CDbl** or CSng to force double-precision or single-precision arithmetic in cases where currency or integer arithmetic normally would occur.

Use the **CDbl** function to provide internationally aware conversions from any other data type to a **Double** subtype. For example, different decimal separators and thousands separators are properly recognized depending on the locale setting of your system.

Data Type Functions

Conversion Functions

Example

This example uses the **CDbl** function to convert an expression to a **Double**.

```
Dim MyCurr, MyDouble
```

```
MyCurr = CCur(234.456784) ' MyCurr is a Currency (234.4567).
' Convert result to a Double (19.2254576).
MyDouble = CDbl(MyCurr * 8.2 * 0.01)
```

2.3.6 CInt Function

Returns an expression that has been converted to a **Variant** of subtype **Integer**.

Syntax

CInt(*Expression***)**

The *Expression* argument is any valid expression.

Remarks

In general, you can document your code using the subtype conversion functions to show that the result of some operation should be expressed as a particular data type rather than the default data type. For example, use **CInt** or CLng to force integer arithmetic in cases where currency, single-precision, or double-precision arithmetic normally would occur.

Use the **CInt** function to provide internationally aware conversions from any other data type to an **Integer** subtype. For example, different decimal separators are properly recognized depending on the locale setting of your system, as are different thousand separators.

If *Expression* lies outside the acceptable range for the **Integer** subtype, an error occurs.

NOTE

CInt differs from the Fix and Int functions, which truncate, rather than round, the fractional part of a number. When the fractional part is exactly 0.5, the **CInt** function always rounds it to the nearest even number. For example, 0.5 rounds to 0, and 1.5 rounds to 2.

Data Type Functions

Conversion Functions

Example

The following example uses the **CInt** function to convert a value to an Integer.

```
Dim MyDouble, MyInt
MyDouble = 2345.5678   ' MyDouble is a Double.
MyInt = CInt( MyDouble) ' MyInt contains 2346.
```

2.3.7 CLng Function

Returns an expression that has been converted to a **Variant** of subtype **Long**.

Syntax

CLng(*Expression***)**

The *Expression* argument is any valid expression.

Remarks

In general, you can document your code using the subtype conversion functions to show that the result of some operation should be expressed as a particular data type rather than the default data type. For example, use CInt or **CLng** to force integer arithmetic in cases where currency, single-precision, or double-precision arithmetic normally would occur.

Use the **CLng** function to provide internationally aware conversions from any other data type to a **Long** subtype. For example, different decimal separators are properly recognized depending on the locale setting of your system, as are different thousand separators.

If *Expression* lies outside the acceptable range for the **Long** subtype, an error occurs.

NOTE

CLng differs from the Fix and Int functions, which truncate, rather than round, the fractional part of a number. When the fractional part is exactly 0.5, the **CLng** function always rounds it to the nearest even number. For example, 0.5 rounds to 0, and 1.5 rounds to 2.

Data Type Functions

Conversion Functions

Example

The following example uses the **CLng** function to convert a value to a **Long**.

```
Dim MyVal1, MyVal2, MyLong1, MyLong2
MyVal1 = 25427.45
MyVal2 = 25427.55      ' MyVal1, MyVal2 are Doubles.
MyLong1 = CLng(MyVal1) ' MyLong1 contains 25427.
MyLong2 = CLng(MyVal2) ' MyLong2 contains 25428.
```

2.3.8 CSng Function

Returns an expression that has been converted to a **Variant** of subtype **Single**.

Syntax

CSng(*Expression***)**

The *Expression* argument is any valid expression.

Remarks

In general, you can document your code using the data type conversion functions to show that the result of some operation should be expressed as a particular data type rather than the default data type. For example, use CDbl or **CSng** to force double-precision or single-precision arithmetic in cases where currency or integer arithmetic normally would occur.

Use the **CSng** function to provide internationally aware conversions from any other data type to a **Single** subtype. For example, different decimal separators are properly recognized depending on the locale setting of your system, as are different thousand separators.

If *Expression* lies outside the acceptable range for the **Single** subtype, an error occurs.


Data Type Functions


Conversion Functions

Example

The following example uses the **CSng** function to convert a value to a **Single**.

```
Dim MyDouble1, MyDouble2    ' MyDouble1, MyDouble2 are Doubles.
Dim MySingle1, MySingle2
MyDouble1 = 75.3421115
MyDouble2 = 75.3421555
MySingle1 = CSng(MyDouble1) ' MySingle1 contains 75.34211.
```

```
MySingle2 = CSng( MyDouble2) '  MySingle2 contains 75.34216.
```

2.3.9 CStr Function

Returns an expression that has been converted to a **Variant** of subtype **String**.

Syntax

CStr(*Expression***)**

The *Expression* argument is any valid expression.

Remarks

In general, you can document your code using the data type conversion functions to show that the result of some operation should be expressed as a particular data type rather than the default data type. For example, use **CStr** to force the result to be expressed as a **String**.

You should use the **CStr** function instead of String to provide internationally aware conversions from any other data type to a **String** subtype. For example, different decimal separators are properly recognized depending on the locale setting of your system.

The data in *Expression* determines what is returned according to the following table.

If *Expression* **is**	**CStr returns**
Boolean	A **String** containing **True** or **False**.
Date	A **String** containing a date in the short-date format of your system.
Null	A run-time error.
Empty	A zero-length **String** ("").
Error	A **String** containing the word Error followed by the error number.
Other numeric	A **String** containing the number.

Data Type Functions

Conversion Functions

Example

The following example uses the **CStr** function to convert a numeric value to a **String**.

```
Dim MyDouble, MyString
MyDouble = 437.324        ' MyDouble is a Double.
MyString = CStr(MyDouble) ' MyString contains "437.324".
```

2.3.10 Fix Function

Returns the integer portion of a number.

Syntax

Fix(*Number***)**

The *Number* argument can be any valid numeric expression. If *Number* contains **Null**, **Null** is returned.

Remarks

Both Int and **Fix** remove the fractional part of *Number* and return the resulting integer value.

The difference between Int and **Fix** is that if *Number* is negative, Int returns the first negative integer less than or equal to *Number,* whereas **Fix** returns the first negative integer greater than or equal to *Number.* For example, Int converts -8.4 to -9, and **Fix** converts -8.4 to -8.

Fix(*Number*) is equivalent to:

```
Sgn(Number) * Int(Abs(Number))
```

Data Type Functions

Conversion Functions

Math Functions

Example

The following examples illustrate how the Int and **Fix** functions return integer portions of numbers.

```
Dim MyNumber
MyNumber = Int(99.8)   ' Returns 99.
MyNumber = Fix(99.2)   ' Returns 99.
MyNumber = Int(-99.8)  ' Returns -100.
MyNumber = Fix(-99.8)  ' Returns -99.
MyNumber = Int(-99.2)  ' Returns -100.
MyNumber = Fix(-99.2)  ' Returns -99.
```

2.3.11 Int Function

Returns the integer portion of a number.

Syntax

Int(*Number***)**

The *Number* argument can be any valid numeric expression. If *Number* contains **Null**, **Null** is returned.

Remarks

Both **Int** and Fix remove the fractional part of *Number* and return the resulting integer value.

The difference between **Int** and Fix is that if *Number* is negative, **Int** returns the first negative integer less than or equal to *Number,* whereas Fix returns the first negative integer greater than or equal to *Number.* For example, **Int** converts -8.4 to -9, and Fix converts -8.4 to -8.

Fix(*Number*) is equivalent to.

```
Sgn(Number) * Int(Abs(Number))
```

Data Type Functions

Conversion Functions

Math Functions

Example

The following examples illustrate how the **Int** and Fix functions return integer portions of numbers.

```
Dim MyNumber
```

```
MyNumber = Int( 99.8)   ' Returns 99.
MyNumber = Fix( 99.2)   ' Returns 99.
MyNumber = Int(-99.8) ' Returns -100.
MyNumber = Fix(-99.8) ' Returns -99.
MyNumber = Int(-99.2) ' Returns -100.
MyNumber = Fix(-99.2) ' Returns -99.
```

2.3.12 IsArray Function

Returns a Boolean value indicating whether a variable is an array.

Syntax

IsArray(*VarName***)**

The *VarName* argument can be any variable.

Remarks

IsArray returns **True** if the variable is an array; otherwise, it returns **False**. **IsArray** is especially useful with variants containing arrays.

Data Type Functions

Array Functions

Example

The following example uses the **IsArray** function to test whether MyVariable is an array.

```
Dim MyVariable
Dim MyArray( 3)
MyArray( 0)  = "Sunday"
MyArray( 1)  = "Monday"
MyArray( 2)  = "Tuesday"
MyVariable = IsArray( MyArray) ' MyVariable contains "True".
```

2.3.13 IsDate Function

Returns a Boolean value indicating whether an expression can be converted to a date.

Syntax

IsDate(*Expression***)**

The *Expression* argument can be any date expression or string expression recognizable as a date or time.

Remarks

IsDate returns **True** if the *Expression* is a date or can be converted to a valid date; otherwise, it returns **False**. In Microsoft Windows, the range of valid dates is January 1, 100 A.D. through December 31, 9999 A.D.; the ranges vary among operating systems.

Data Type Functions

Example

The following example uses the **IsDate** function to determine whether an expression can be converted to a date.

```
Dim MyDate, YourDate, NoDate, MyCheck
MyDate = "October 19, 1962"
YourDate = #10/19/62#
NoDate = "Hello"
MyCheck = IsDate( MyDate)    ' Returns True.
MyCheck = IsDate( YourDate) ' Returns True.
MyCheck = IsDate( NoDate)    ' Returns False.
```

2.3.14 IsEmpty Function

Returns a Boolean value indicating whether a variable has been initialized.

Syntax

IsEmpty(*Expression***)**

The *Expression* argument can be any expression. However, because **IsEmpty** is used to determine if individual variables are initialized, the *Expression* argument is most often a single variable name.

Remarks

IsEmpty returns **True** if the variable is uninitialized, or is explicitly set to **Empty**; otherwise, it returns **False**. **False** is always returned if *Expression* contains more than one variable.

Data Type Functions

Example

The following example uses the **IsEmpty** function to determine whether a variable has been initialized.

```
Dim MyVar, MyCheck
MyCheck = IsEmpty( MyVar) ' Returns True.
MyVar = Null              ' Assign Null.
MyCheck = IsEmpty( MyVar) ' Returns False.
MyVar = Empty             ' Assign Empty.
MyCheck = IsEmpty( MyVar) ' Returns True.
```

2.3.15 IsNull Function

Returns a Boolean value that indicates whether an expression contains no valid data (**Null**).

Syntax

IsNull(*Expression***)**

The *Expression* argument can be any expression.

Remarks

IsNull returns **True** if *Expression* is **Null**, that is, it contains no valid data; otherwise, **IsNull** returns **False**. If *Expression* consists of more than one variable, **Null** in any constituent variable causes **True** to be returned for the entire expression.

The **Null** value indicates that the variable contains no valid data. **Null** is not the same as **Empty**, which indicates that a variable has not yet been initialized. It is also not the same as a zero-length string (""), which is sometimes referred to as a null string.

IMPORTANT

Use the **IsNull** function to determine whether an expression contains a **Null** value. Expressions that you might expect to evaluate to **True** under some circumstances, such as If Var = Null and If Var <> Null, are always **False**. This is because any expression containing a **Null** is itself **Null**, and therefore, **False**.

Data Type Functions

Example

The following example uses the **IsNull** function to determine whether a variable contains a **Null**.

```
Dim MyVar, MyCheck
MyCheck = IsNull(MyVar) ' Returns False.
MyVar = Null            ' Assign Null.
MyCheck = IsNull(MyVar) ' Returns True.
MyVar = Empty           ' Assign Empty.
MyCheck = IsNull(MyVar) ' Returns False.
```

2.3.16 IsNumeric Function

Returns a Boolean value indicating whether an expression can be evaluated as a number.

Syntax

IsNumeric(*Expression***)**

The *Expression* argument can be any expression.

Remarks

IsNumeric returns **True** if the entire *Expression* is recognized as a number; otherwise, it returns **False**. **IsNumeric** returns **False** if *Expression* is a date expression.

Data Type Functions

Example

The following example uses the **IsNumeric** function to determine whether a variable can be evaluated as a number.

```
Dim MyVar, MyCheck
MyVar = 53                  ' Assign a value.
MyCheck = IsNumeric(MyVar) ' Returns True.
MyVar = "459.95"            ' Assign a value.
MyCheck = IsNumeric(MyVar) ' Returns True.
MyVar = "45 Help"           ' Assign a value.
MyCheck = IsNumeric(MyVar) ' Returns False.
```

2.3.17 IsObject Function

Returns a Boolean value indicating whether an expression references a valid Automation object.

Syntax

IsObject(*Expression***)**

The *Expression* argument can be any expression.

Remarks

IsObject returns **True** if *Expression* is a variable of **Object** subtype or a user-defined object; otherwise, it returns **False**.

Data Type Functions

Example

The following example uses the **IsObject** function to determine if an identifier represents an object variable.

```
Dim MyInt, MyCheck, MyObject
Set MyObject = Me
MyCheck = IsObject( MyObject) ' Returns True.
MyCheck = IsObject( MyInt)    ' Returns False.
```

2.3.18 TypeName Function

Returns a string that provides **Variant** subtype information about a variable.

Syntax

TypeName(*VarName***)**

The required *VarName* argument can be any variable.

Return Values

The **TypeName** function has the following return values:

Value	Description
Byte	Byte value.
Integer	Integer value.
Long	Long integer value.
Single	Single-precision floating-point value.
Double	Double-precision floating-point value.
Currency	Currency value.
Decimal	Decimal value.
Date	Date or time value.
String	Character string value.
Boolean	Boolean value; **True** or **False**.
Empty	Unitialized.
Null	No valid data.
<Object type>	Actual type name of an object.
Object	Generic object.
Unknown	Unknown object type.
Nothing	Object variable that doesn't yet refer to an object instance.
Error	Error.

Data Type Functions

Example

The following example uses the **TypeName** function to return information about a variable.

```
Dim ArrayVar(4), MyType
NullVar = Null                  ' Assign Null value.
MyType = TypeName("VBScript") ' Returns "String".
MyType = TypeName(4)            ' Returns "Integer".
MyType = TypeName(37.50)        ' Returns "Double".
MyType = TypeName(NullVar)      ' Returns "Null".
MyType = TypeName(ArrayVar)     ' Returns "Variant()".
```

2.3.19 VarType Function

Returns a value indicating the subtype of a variable.

Syntax

VarType(*VarName***)**

The *VarName* argument can be any variable.

Return Values

The **VarType** function returns the following values:

Constant	Value	Description
vbEmpty	0	**Empty** (uninitialized).
vbNull	1	**Null** (no valid data).
vbInteger	2	Integer.
vbLong	3	Long integer.
vbSingle	4	Single-precision floating-point number.
vbDouble	5	Double-precision floating-point number.

vbCurrency	6	Currency.
vbDate	7	Date.
vbString	8	String.
vbObject	9	Automation object.
vbError	10	Error.
vbBoolean	11	Boolean.
vbVariant	12	Variant (used only with arrays of Variants).
vbDataObject	13	A data-access object.
vbByte	17	Byte.
vbArray	8192	Array.

Remarks

The **VarType** function never returns the value for Array by itself. It is always added to some other value to indicate an array of a particular type. The value for Variant is only returned when it has been added to the value for Array to indicate that the argument to the **VarType** function is an array. For example, the value returned for an array of integers is calculated as 2 + 8192, or 8194. If an object has a default property, **VarType (***Object***)** returns the type of its default property.

NOTE

These constants are specified by *VBScript*. As a result, the names can be used anywhere in your code in place of the actual values.

Data Type Functions

Example

The following example uses the **VarType** function to determine the subtype of a variable.

```
Dim MyCheck
MyCheck = VarType(300)          ' Returns 2.
MyCheck = VarType(#10/19/62#) ' Returns 7.
MyCheck = VarType("VBScript") ' Returns 8.
```

2.4 Date and Time Functions

- **Date** Function
- **DateAdd** Function
- **DateDiff** Function
- **DatePart** Function
- **DateSerial** Function
- **DateValue** Function
- **Day** Function
- **Hour** Function
- **Minute** Function
- **Month** Function
- **MonthName** Function
- **Now** Function
- **Second** Function
- **Time** Function
- **Timer** Function
- **TimeSerial** Function
- **TimeValue** Function
- **WeekDay** Function
- **WeekDayName** Function
- **Year** Function

2.4.1 Date Function

Returns the current system date.

Syntax

Date

Date and Time Functions

Example

The following example uses the **Date** function to return the current system date.

```
Dim MyDate
MyDate = Date '  MyDate contains the current system date.
```

2.4.2 DateAdd Function

Returns a date to which a specified time interval has been added.

Syntax

DateAdd(*Interval***,** *Number***,** *Date***)**

The **DateAdd** function syntax has these parts:

Part	Description
Interval	Required. String expression that is the interval you want to add. See Settings section for values.
Number	Required. Numeric expression that is the number of interval you want to add. The numeric expression can either be positive, for dates in the future, or negative, for dates in the past.
Date	Required. **Variant** or literal representing the date to which *Interval* is added.

Settings

The *Interval* argument can have the following values:

Setting	Description
yyyy	Year
q	Quarter

m	Month
y	Day of year
d	Day
w	Weekday
ww	Week of year
h	Hour
n	Minute
s	Second

Remarks

You can use the **DateAdd** function to add or subtract a specified time interval from a date. For example, you can use **DateAdd** to calculate a date 30 days from today or a time 45 minutes from now. To add days to *Date*, you can use Day of Year ("y"), Day ("d"), or Weekday ("w").

The **DateAdd** function won't return an invalid date. The following example adds one month to January 31.

```
NewDate = DateAdd("m", 1, "31-Jan-95")
```

In this case, **DateAdd** returns 28-Feb-95, not 31-Feb-95. If *Date* is 31-Jan-96, it returns 29-Feb-96 because 1996 is a leap year.

If the calculated date would precede the year 100, an error occurs.

If number isn't a **Long** value, it is rounded to the nearest whole number before being evaluated.

Date and Time Functions

2.4.3 DateDiff Function

Returns the number of intervals between two dates.

Syntax

DateDiff(*Interval*, *Date1*, *Date2***[,** *FirstDayOfWeek***[,** *FirstWeekOfYear***]])**

The **DateDiff** function syntax has these parts:

Part	Description
Interval	Required. String expression that is the interval you want to use to calculate the differences between *Date1* and *Date2*. See Settings section for values.
Date1, Date2	Required. Date expressions. Two dates you want to use in the calculation.
FirstDayOfWeek	Optional. Constant that specifies the day of the week. If not specified, Sunday is assumed. See Settings section for values.
FirstWeekOfYear	Optional. Constant that specifies the first week of the year. If not specified, the first week is assumed to be the week in which January 1 occurs. See Settings section for values.

Settings

The *Interval* argument can have the following values:

Setting	Description
yyyy	Year
q	Quarter

m	Month
y	Day of year
d	Day
w	Weekday
ww	Week of year
h	Hour
n	Minute
s	Second

The *FirstDayOfWeek* argument can have the following values:

Constant	Value	Description
vbUseSystem	0	Use National Language Support (NLS) API setting.
vbSunday	1	Sunday (default)
vbMonday	2	Monday
vbTuesday	3	Tuesday
vbWednesday	4	Wednesday
vbThursday	5	Thursday
vbFriday	6	Friday
vbSaturday	7	Saturday

The *FirstWeekOfYear* argument can have the following values:

Constant	Value	Description
vbUseSystem	0	Use National Language Support (NLS) API setting.
vbFirstJan1	1	Start with the week in which January 1 occurs (default).
vbFirstFourDays	2	Start with the week that has at least four days in the new year.
vbFirstFullWeek	3	Start with the first full weekof the new year.

Remarks

You can use the **DateDiff** function to determine how many specified time intervals exist between two dates. For example, you might use **DateDiff** to calculate the number of days between two dates, or the number of weeks between today and the end of the year.

To calculate the number of days between *Date1* and *Date2*, you can use either Day of year ("y") or Day ("d"). When *Interval* is Weekday ("w"), **DateDiff** returns the number of weeks between the two dates. If *Date1* falls on a Monday, **DateDiff** counts the number of Mondays until *Date2*. It counts *Date2* but not *Date1*. If *Interval* is Week ("ww"), however, the **DateDiff** function returns the number of calendar weeks between the two dates. It counts the number of Sundays between *Date1* and *Date2*. **DateDiff** counts *Date2* if it falls on a Sunday; but it doesn't count *Date1*, even if it does fall on a Sunday.

If *Date1* refers to a later point in time than *Date2*, the **DateDiff** function returns a negative number.

The *FirstDayOfWeek* argument affects calculations that use the "w" and "ww" interval symbols.

If *Date1* or *Date2* is a date literal, the specified year becomes a permanent part of that date. However, if *Date1* or *Date2* is enclosed in quotation marks (" ") and you omit the year, the current year is inserted in your code each time the *Date1* or *Date2* expression is evaluated. This makes it possible to write code that can be used in different years.

When comparing December 31 to January 1 of the immediately succeeding year, **DateDiff** for Year ("yyyy") returns 1 even though only a day has elapsed.

Date and Time Functions

Example

The following example uses the **DateDiff** function to display the number of days between a given date and today.

```
Function DiffADate( TheDate)
  DiffADate = "Days from today: " & DateDiff( "d", Now, TheDate)
End Function
```

2.4.4 DatePart Function

Returns the specified part of a given date.

Syntax

DatePart(*Interval***,** *Date***[,** *FirstDayOfWeek***[,** *FirstWeekOfYear***]])**

The **DatePart** function syntax has these parts:

Part	Description
Interval	Required. String expression that is the interval of time you want to return. See Settings section for values.
Date	Required. Date expression you want to evaluate.
FirstDayofWeek	Optional. Constant that specifies the day of the week. If not specified, Sunday is assumed. See Settings section for values.
FirstWeekOfYear	Optional. Constant that specifies the first week of the year. If not specified, the first week is assumed to be the week in which January 1 occurs. See Settings section for values.

Settings

The *Interval* argument can have the following values:

Setting	Description
yyyy	Year
q	Quarter
m	Month
y	Day of year
d	Day
w	Weekday
ww	Week of year
h	Hour
n	Minute
s	Second

The *FirstDayOfWeek* argument can have the following values:

Constant	Value	Description
vbUseSystem	0	Use National Language Support (NLS) API setting.
vbSunday	1	Sunday (default)
vbMonday	2	Monday
vbTuesday	3	Tuesday
vbWednesday	4	Wednesday
vbThursday	5	Thursday
vbFriday	6	Friday
vbSaturday	7	Saturday

The *FirstWeekOfYear* argument can have the following values:

Constant	Value	Description
vbUseSystem	0	Use National Language Support (NLS) API setting.
vbFirstJan1	1	Start with the week in which January 1 occurs (default).
vbFirstFourDays	2	Start with the week that has at least four days in the new year.
vbFirstFullWeek	3	Start with the first full weekof the new year.

Remarks

You can use the **DatePart** function to evaluate a date and return a specific interval of time. For example, you might use **DatePart** to calculate the day of the week or the current hour.

The *FirstDayOfWeek* argument affects calculations that use the "w" and "ww" interval symbols.

If *Date* is a date literal, the specified year becomes a permanent part of that date. However, if *Date* is enclosed in quotation marks (" "), and you omit the year, the current year is inserted in your code each time the *Date* expression is evaluated. This makes it possible to write code that can be used in different years.

Date and Time Functions

Example

This example takes a date and, using the **DatePart** function, displays the quarter of the year in which it occurs.

```
Function GetQuarter( TheDate)
  GetQuarter = DatePart( "q", TheDate)
End Function
```

2.4.5 DateSerial Function

Returns a **Variant** of subtype **Date** for a specified year, month, and day.

Syntax

DateSerial(*Year***,** *Month***,** *Day***)**

The **DateSerial** function syntax has these parts:

Part	Description
Year	Number between 100 and 9999, inclusive, or a numeric expression.
Month	Any numeric expression.
Day	Any numeric expression.

Remarks

To specify a date, such as December 31, 1991, the range of numbers for each **DateSerial** argument should be in the accepted range for the unit; that is, 1–31 for days and 1–12 for months. However, you can also specify relative dates for each argument using any numeric expression that represents some number of days, months, or years before or after a certain date.

For the *Year* argument, values between 0 and 99, inclusive, are interpreted as the years 1900–1999. For all other *Year* arguments, use a complete four-digit year (for example, 1800).

When any argument exceeds the accepted range for that argument, it increments to the next larger unit as appropriate. For example, if you specify 35 days, it is evaluated as one month and some number of days, depending on where in the year it is applied. However, if any single argument is outside the range -32,768 to 32,767, or if the date specified by the three arguments, either directly or by expression, falls outside the acceptable range of dates, an error occurs.

Date and Time Functions

Conversion Functions

Example

The following example uses numeric expressions instead of absolute date numbers. Here the **DateSerial** function returns a date that is the day before the first day (1 - 1) of two months before August (8 - 2) of 10 years before 1990 (1990 - 10); in other words, May 31, 1980.

```
Dim MyDate1, MyDate2
MyDate1 = DateSerial(1970, 1, 1)              ' Returns January 1, 1970.
MyDate2 = DateSerial(1990 - 10, 8 - 2, 1 - 1) ' Returns May 31, 1980.
```

2.4.6 DateValue Function

Returns a **Variant** of subtype **Date**.

Syntax

DateValue(*Date***)**

The *Date* argument is normally a string expression representing a date from January 1, 100 through December 31, 9999. However, *Date* can also be any expression that can represent a date, a time, or both a date and time, in that range.

Remarks

If the *Date* argument includes time information, **DateValue** doesn't return it. However, if *Date* includes invalid time information (such as "89:98"), an error occurs.

If *Date* is a string that includes only numbers separated by valid date separators, **DateValue** recognizes the order for month, day, and year according to the short date format you specified for your system. **DateValue** also recognizes unambiguous dates that contain month names, either in long or abbreviated form. For example, in addition to recognizing 12/30/1991 and 12/30/91, **DateValue** also recognizes December 30, 1991 and Dec 30, 1991.

If the year part of *Date* is omitted, **DateValue** uses the current year from your computer's system date.

Date and Time Functions

Conversion Functions

Example

The following example uses the **DateValue** function to convert a string to a date. You can also use date literals to directly assign a date to a **Variant** variable, for example, MyDate = #9/11/63#.

```
Dim MyDate
MyDate = DateValue("September 11, 1963") ' Return a date.
```

2.4.7 Day Function

Returns a whole number between 1 and 31, inclusive, representing the day of the month.

Syntax

Day(*Date***)**

The *Date* argument is any expression that can represent a date. If *Date* contains **Null**, **Null** is returned.

Date and Time Functions

Conversion Functions

Example

The following example uses the **Day** function to obtain the day of the month from a specified date.

```
Dim MyDay
MyDay = Day("October 19, 1962") ' MyDay contains 19.
```

2.4.8 Hour Function

Returns a whole number between 0 and 23, inclusive, representing the hour of the day.

Syntax

Hour(*Time***)**

The *Time* argument is any expression that can represent a time. If *Time* contains **Null**, **Null** is returned.

Date and Time Functions
Conversion Functions

Example

The following example uses the **Hour** function to obtain the hour from the current time.

```
Dim MyTime, MyHour
MyTime = Now
' MyHour contains the number representing the current hour.
MyHour = Hour( MyTime)
```

2.4.9 Minute Function

Returns a whole number between 0 and 59, inclusive, representing the minute of the hour.

Syntax

Minute(*Time***)**

The *Time* argument is any expression that can represent a time. If *Time* contains **Null**, **Null** is returned.

Date and Time Functions
Conversion Functions

Example

The following example uses the **Minute** function to return the minute of the hour.

```
Dim MyVar
MyVar = Minute( Now)
```

2.4.10 Month Function

Returns a whole number between 1 and 12, inclusive, representing the month of the year.

Syntax

Month(*Date***)**

The *Date* argument is any expression that can represent a date. If *Date* contains **Null**, **Null** is returned.

Date and Time Functions

Conversion Functions

Example

The following example uses the **Month** function to return the current month.

```
Dim MyVar
' MyVar contains the number corresponding to the current month.
MyVar = Month( Now)
```

2.4.11 MonthName Function

Returns a string indicating the specified month.

Syntax

MonthName(*Month[***, *Abbreviate]***)**

The **MonthName** function syntax has these parts:

Part	Description
Month	Required. The numeric designation of the month. For example, January is 1, February is 2, and so on.
Abbreviate	Optional. Boolean value that indicates if the month name is to be abbreviated. If omitted, the default is **False**, which means that the month name is not abbreviated.

Date and Time Functions

String Functions

Example

The following example uses the **MonthName** function to return an abbreviated month name for a date expression.

```
Dim MyVar
MyVar = MonthName(10, True) ' MyVar contains "Oct".
```

2.4.12 Now Function

Returns the current date and time according to the setting of your computer's system date and time.

Syntax

Now

Date and Time Functions

Example

The following example uses the **Now** function to return the current date and time.

```
Dim MyVar
MyVar = Now ' MyVar contains the current date and time.
```

2.4.13 Second Function

Returns a whole number between 0 and 59, inclusive, representing the second of the minute.

Syntax

Second(*Time***)**

The *Time* argument is any expression that can represent a time. If *Time* contains **Null**, **Null** is returned.

Date and Time Functions

Conversion Functions

Example

The following example uses the **Second** function to return the current second.

```
Dim MySec
' MySec contains the number representing the current second.
MySec = Second( Now)
```

2.4.14 Time Function

Returns a **Variant** of subtype **Date** indicating the current system time.

Syntax

```
Time
```

 Date and Time Functions

Example

The following example uses the **Time** function to return the current system time.

```
Dim MyTime
MyTime = Time ' Return current system time.
```

2.4.15 Timer Function

Returns the number of seconds that have elapsed since 12:00 AM (midnight).

Syntax

```
Timer
```

 Date and Time Functions

Example

The following example uses the **Timer** function to determine the time it takes to iterate a For...Next loop *N* times.

```
Function TimeIt( N)
  Dim StartTime, EndTime
  StartTime = Timer
  For I = 1 To N
  Next
  EndTime = Timer
  TimeIt = EndTime - StartTime
End Function
```

2.4.16 TimeSerial Function

Returns a **Variant** of subtype **Date** containing the time for a specific hour, minute, and second.

Syntax

TimeSerial(*Hour***,** *Minute***,** *Second***)**

The **TimeSerial** function syntax has these parts:

Part	Description
Hour	Number between 0 (12:00 A.M.) and 23 (11:00 P. M.), inclusive, or a numeric expression.
Minute	Any numeric expression.
Second	Any numeric expression.

Remarks

To specify a time, such as 11:59:59, the range of numbers for each **TimeSerial** argument should be in the accepted range for the unit; that is, 0–23 for hours and 0–59 for minutes and seconds. However, you can also specify relative times for each argument using any numeric expression that represents some number of hours, minutes, or seconds before or after a certain time.

When any argument exceeds the accepted range for that argument, it increments to the next larger unit as appropriate. For example, if you specify 75 minutes, it is evaluated as one hour and 15 minutes. However, if any single argument is outside the range -32,768 to 32,767, or if the time specified by the three arguments, either directly or by expression, causes the date to fall outside the acceptable range of dates, an error occurs.

Date and Time Functions

Conversion Functions

Example

The following example uses expressions instead of absolute time numbers. The **TimeSerial** function returns a time for 15 minutes before (-15) six hours before noon (12 - 6), or 5:45:00 A.M.

```
Dim MyTime
MyTime = TimeSerial(12 - 6, -15, 0) ' Returns 5:45:00 AM.
```

2.4.17 TimeValue Function

Returns a **Variant** of subtype **Date** containing the time.

Syntax

TimeValue(*Time***)**

The *Time* argument is usually a string expression representing a time from 0:00:00 (12:00:00 A.M.) to 23:59:59 (11:59:59 P.M.), inclusive. However, *Time* can also be any expression that represents a time in that range. If *Time* contains **Null**, **Null** is returned.

Remarks

You can enter valid times using a 12-hour or 24-hour clock. For example, "2:24PM" and "14:24" are both valid *Time* arguments. If the *Time* argument contains date information, **TimeValue** doesn't return the date information. However, if *Time* includes invalid date information, an error occurs.

Date and Time Functions

Conversion Functions

Example

The following example uses the **TimeValue** function to convert a string to a time. You can also use date literals to directly assign a time to a **Variant** (for example, MyTime = #4:35:17 PM#).

```
Dim MyTime
MyTime = TimeValue("4:35:17 PM") ' MyTime contains 4:35:17 PM.
```

2.4.18 Weekday Function

Returns a whole number representing the day of the week.

Syntax

Weekday(*Date***[, ***FirstDayOfWeek***])**

The **Weekday** function syntax has these parts:

Part	Description
Date	Any expression that can represent a date. If *Date* contains **Null**, **Null** is returned.
FirstDayOfWeek	A constant that specifies the first day of the week. If omitted, **vbSunday** is assumed.

Settings

The *FirstDayOfWeek* argument has these settings:

Constant	Value	Description
vbUseSystem	0	Use National Language Support (NLS) API setting.
vbSunday	1	Sunday
vbMonday	2	Monday
vbTuesday	3	Tuesday
vbWednesday	4	Wednesday
vbThursday	5	Thursday
vbFriday	6	Friday
vbSaturday	7	Saturday

Return Values

The **Weekday** function can return any of these values:

Constant	Value	Description
vbSunday	1	Sunday
vbMonday	2	Monday
vbTuesday	3	Tuesday
vbWednesday	4	Wednesday
vbThursday	5	Thursday
vbFriday	6	Friday
vbSaturday	7	Saturday

Date and Time Functions

Conversion Functions

Example

The following example uses the **Weekday** function to obtain the day of the week from a specified date.

```
Dim MyDate, MyWeekDay
MyDate = #October 19, 1962# ' Assign a date.
' MyWeekDay contains 6 because MyDate represents a Friday.
MyWeekDay = Weekday( MyDate)
```

2.4.19 WeekdayName Function

Returns a string indicating the specified day of the week.

Syntax

WeekdayName(*WeekDay[, Abbreviate[***, ***FirstDayOfWeek]]***)**

The **WeekdayName** function syntax has these parts:

Part	Description
WeekDay	Required. The numeric designation for the day of the week. Numeric value of each day depends on setting of the *FirstDayOfWeek* setting.
Abbreviate	Optional. Boolean value that indicates if the weekday name is to be abbreviated. If omitted, the default is **False**, which means that the weekday name is not abbreviated.
FirstDayOfWeek	Optional. Numeric value indicating the first day of the week. See Settings section for values.

Settings

The *FirstDayOfWeek* argument can have the following values:

Constant	Value	Description
vbUseSystem	0	Use National Language Support (NLS) API setting.
vbSunday	1	Sunday
vbMonday	2	Monday
vbTuesday	3	Tuesday
vbWednesday	4	Wednesday
vbThursday	5	Thursday
vbFriday	6	Friday
vbSaturday	7	Saturday

Date and Time Functions

String Functions

Example

The following example uses the **WeekDayName** function to return the specified day.

```
Dim MyDate
MyDate = WeekDayName( 6, True) '  MyDate contains Fri.
```

2.4.20 Year Function

Returns a whole number representing the year.

Syntax

Year(*Date***)**

The *Date* argument is any expression that can represent a date. If *Date* contains **Null**, **Null** is returned.

Date and Time Functions

Conversion Functions

Example

The following example uses the **Year** function to obtain the year from a specified date.

```
Dim MyDate, MyYear
MyDate = #October 19, 1962# '  Assign a date.
MyYear = Year( MyDate)       '  MyYear contains 1962.
```

2.5 Array Functions

- **Array** Function
- **Dim** Statement
- **Erase** Statement
- **Filter** Function
- **IsArray** Function
- **Join** Function
- **LBound** Function
- **Private** Statement
- **Public** Statement
- **ReDim** Statement
- **Split** Function
- **UBound** Function

2.5.1 Array Function

Returns a **Variant** containing an array.

Syntax

Array(*ArgList***)**

The required *ArgList* argument is a comma-delimited list of values that are assigned to the elements of an array contained with the **Variant**. If no arguments are specified, an array of zero length is created.

Remarks

The notation used to refer to an element of an array consists of the variable name followed by parentheses containing an index number indicating the desired element.

NOTE

A variable that is not declared as an array can still contain an array. Although a **Variant** variable containing an array is conceptually different from an array variable containing **Variant** elements, the array elements are accessed in the same way.

Array Functions

Example

In the following example, the first statement creates a variable named A. The second statement assigns an array to variable A. The last statement assigns the value contained in the second array element to another variable.

```
Dim A
A = Array(10, 20, 30)
B = A(2) '  B is now 30.
```

2.5.2 Dim Statement

Declares variables and allocates storage space.

Syntax

Dim *VarName*[**(**[*Subscripts*]**)**][**,** *VarName*[**(**[*Subscripts*]**)**]]...

The **Dim** statement syntax has these parts:

Part	Description
VarName	Name of the variable; follows standard variable naming conventions.
Subscripts	Dimensions of an array variable; up to 60 multiple dimensions may be declared. The *Subscripts* argument uses the following syntax: *upperbound*[, *upperbound*]... The lower bound of an array is always zero.

Remarks

Variables declared with **Dim** at the script level (globally) are available to all procedures in all scripts within the same thread (similar to **Public** statement). At the procedure level, variables are available only within the procedure.

You can also use the **Dim** statement with empty parentheses to declare a dynamic array. After declaring a dynamic array, use the ReDim statement within a procedure to define the number of dimensions and elements in the array. If you try to redeclare a dimension for an array variable whose size was explicitly specified in a **Dim** statement, an error occurs.

TIP

When you use the **Dim** statement in a procedure, you generally put the **Dim** statement at the beginning of the procedure.

Array Functions

VBScript Statements

Example

The following examples illustrate the use of the **Dim** statement.

```
Dim Names(9)      ' Declare an array with 10 elements.
Dim Names()       ' Declare a dynamic array.
Dim MyVar, MyNum ' Declare two variables.
```

2.5.3 Erase Statement

Reinitializes the elements of fixed-size arrays and deallocates dynamic-array storage space.

Syntax

Erase *Array*

The *Array* argument is the name of the array variable to be erased.

Remarks

It is important to know whether an array is fixed-size (ordinary) or dynamic because **Erase** behaves differently depending on the type of array. **Erase** recovers no memory for fixed-size arrays. **Erase** sets the elements of a fixed array as follows:

Type of array	Effect of Erase on fixed-array elements
Fixed numeric array	Sets each element to zero.

Fixed string array	Sets each element to zero-length ("").
Array of objects	Sets each element to the special value **Nothing**.

Erase frees the memory used by dynamic arrays. Before your program can refer to the dynamic array again, it must redeclare the array variable's dimensions using a ReDim statement.

Array Functions

VBScript Statements

Example

The following example illustrates the use of the **Erase** statement.

```
Dim NumArray(9)
Dim DynamicArray()
ReDim DynamicArray(9) ' Allocate storage space.
Erase NumArray        ' Each element is reinitialized.
Erase DynamicArray    ' Free memory used by array.
```

2.5.4 Filter Function

Returns a zero-based array containing a subset of a string array based on a specified filter criteria.

Syntax

Filter(*InputStrings***,** *Value***[,** *Include***[,** *Compare***]])**

The **Filter** function syntax has these parts:

Part	Description
InputStrings	Required. One-dimensional array of strings to be searched.
Value	Required. String to search for.

Include	Optional. Boolean value indicating whether to return substrings that include or exclude *Value*. If *Include* is **True**, **Filter** returns the subset of the array that contains *Value* as a substring. If *Include* is **False**, **Filter** returns the subset of the array that does not contain *Value* as a substring.
Compare	Optional. Numeric value indicating the kind of string comparison to use. See Settings section for values.

Settings

The *Compare* argument can have the following values:

Constant	Value	Description
vbBinaryCompare	0	Perform a binary comparison.
vbTextCompare	1	Perform a textual comparison.

Remarks

If no matches of *Value* are found within *InputStrings*, **Filter** returns an empty array. An error occurs if *InputStrings* is **Null** or is not a one-dimensional array.

The array returned by the **Filter** function contains only enough elements to contain the number of matched items.

Array Functions

Example

The following example uses the **Filter** function to return the array containing the search criteria "Mon".

```
Dim MyIndex
Dim MyArray (3)
MyArray(0) = "Sunday"
```

```
MyArray(1) = "Monday"
MyArray(2) = "Tuesday"
MyIndex = Filter(MyArray, "Mon") ' MyIndex(0) contains "Monday".
```

2.5.5 IsArray Function

Returns a Boolean value indicating whether a variable is an array.

Syntax

IsArray(*VarName***)**

The *VarName* argument can be any variable.

Remarks

IsArray returns **True** if the variable is an array; otherwise, it returns **False**. **IsArray** is especially useful with variants containing arrays.

Data Type Functions

Array Functions

Example

The following example uses the **IsArray** function to test whether MyVariable is an array.

```
Dim MyVariable
Dim MyArray(3)
MyArray(0) = "Sunday"
MyArray(1) = "Monday"
MyArray(2) = "Tuesday"
MyVariable = IsArray(MyArray) ' MyVariable contains "True".
```

2.5.6 Join Function

Returns a string created by joining a number of substrings contained in an array.

Syntax

Join(*List***[, *Delimiter*])**

The **Join** function syntax has these parts:

Part	Description
List	Required. One-dimensional array containing substrings to be joined.
Delimiter	Optional. String character used to separate the substrings in the returned string. If omitted, the space character (" ") is used. If *Delimiter* is a zero-length string, all items in the list are concatenated with no delimiters.

Array Functions

Example

The following example uses the **Join** function to join the substrings of MyArray.

```
Dim MyString
Dim MyArray( 4)
MyArray( 0)  = "Mr."
MyArray( 1)  = "John"
MyArray( 2)  = "Doe"
MyArray( 3)  = "III"
MyString = Join( MyArray)  '  MyString contains "Mr. John Doe III".
```

2.5.7 LBound Function

Returns the smallest available subscript for the indicated dimension of an array.

Syntax

LBound(*ArrayName***[,** *Dimension***])**

The **LBound** function syntax has these parts:

Part	Description
ArrayName	Name of the array variable; follows standard variable naming conventions.

Dimension	Whole number indicating which dimension's lower bound is returned. Use 1 for the first dimension, 2 for the second, and so on. If *Dimension* is omitted, 1 is assumed.

Remarks

The **LBound** function is used with the UBound function to determine the size of an array. Use the UBound function to find the upper limit of an array dimension.

The lower bound for any dimension is always 0.

Array Functions

2.5.8 Private Statement

Declares private variables and allocates storage space. Declares, in a Class block, a private variable.

Syntax

`Private` *VarName*[**(**[*Subscripts*]**)**][**,** *VarName*[**(**[*Subscripts*]**)**]]...

The **Private** statement syntax has these parts:

Part	Description
VarName	Name of the variable; follows standard variable naming conventions.
Subscripts	Dimensions of an array variable; up to 60 multiple dimensions may be declared. The *Subscripts* argument uses the following syntax: *upper*[, *upper*]... The lower bound of an array is always zero.

Remarks

A variable declared with the **Private** statement at a script level (globally) is available in all scripts within the same thread (similar to the **Public** statement). The **Private** statement can still be meaningfully used in a declaration of objects (with the **Class** statement).

A variable that refers to an object must be assigned an existing object using the Set statement before it can be used. Until it is assigned an object, the declared object variable is initialized as **Empty**.

You can also use the **Private** statement with empty parentheses to declare a dynamic array. After declaring a dynamic array, use the ReDim statement within a procedure to define the number of dimensions and elements in the array. If you try to redeclare a dimension for an array variable whose size was explicitly specified in a **Private**, Public, or Dim statement, an error occurs.

Array Functions

VBScript Statements

Example

The following example illustrates use of the **Private** statement.

```
Private MyNumber   ' Private Variant variable.
Private MyArray(9) ' Private array variable.
' Multiple Private declarations of Variant variables.
Private MyNumber, MyVar, YourNumber
```

2.5.9 Public Statement

Declares public variables and allocates storage space. Declares, in a Class block, a private variable.

Syntax

Public *VarName*[**(**[*Subscripts*]**)**][**,** *VarName*[**(**[*Subscripts*]**)**]]...

The **Public** statement syntax has these parts:

Part	Description

VarName	Name of the variable; follows standard variable naming conventions.
Subscripts	Dimensions of an array variable; up to 60 multiple dimensions may be declared. The *Subscripts* argument uses the following syntax: `upper[, upper]...` The lower bound of an array is always zero.

Remarks

Public statement variables are available to all procedures in all scripts.

A variable that refers to an object must be assigned an existing object using the Set statement before it can be used. Until it is assigned an object, the declared object variable is initialized as **Empty**.

You can also use the **Public** statement with empty parentheses to declare a dynamic array. After declaring a dynamic array, use the ReDim statement within a procedure to define the number of dimensions and elements in the array. If you try to redeclare a dimension for an array variable whose size was explicitly specified in a Private, **Public**, or Dim statement, an error occurs.

Array Functions

VBScript Statements

Example

The following example illustrates the use of the **Public** statement.

```
Public MyNumber   ' Public Variant variable.
Public MyArray(9) ' Public array variable.
' Multiple Public declarations of Variant variables.
Public MyNumber, MyVar, YourNumber
```

2.5.10 ReDim Statement

Declares dynamic-array variables, and allocates or reallocates storage space at procedure

level.

Syntax

ReDim *[***Preserve***] VarName***(***Subscripts***)***[***, VarName***(***Subscripts***)***]...***

The **ReDim** statement syntax has these parts:

Part	Description
Preserve	Preserves the data in an existing array when you change the size of the last dimension.
VarName	Name of the variable; follows standard variable naming conventions.
Subscripts	Dimensions of an array variable; up to 60 multiple dimensions may be declared. The *Subscripts* argument uses the following syntax: *upper[, upper]...* The lower bound of an array is always zero.

Remarks

The **ReDim** statement is used to size or resize a dynamic array that has already been formally declared using a Private, Public, or Dim statement with empty parentheses (without dimension subscripts). You can use the **ReDim** statement repeatedly to change the number of elements and dimensions in an array.

If you use the **Preserve** keyword, you can resize only the last array dimension, and you can't change the number of dimensions at all. For example, if your array has only one dimension, you can resize that dimension because it is the last and only dimension. However, if your array has two or more dimensions, you can change the size of only the last dimension and still preserve the contents of the array.

The following example shows how you can increase the size of the last dimension of a dynamic array without erasing any existing data contained in the array.

```
ReDim X(10, 10, 10)
  '...
ReDim Preserve X(10, 10, 15)
```

CAUTION

If you make an array smaller than it was originally, data in the eliminated elements is lost.

When variables are initialized, a numeric variable is initialized to 0 and a string variable is initialized to a zero-length string (""). A variable that refers to an object must be assigned an existing object using the Set statement before it can be used. Until it is assigned an object, the declared object variable has the special value **Nothing**.

Array Functions

VBScript Statements

2.5.11 Split Function

Returns a zero-based, one-dimensional array containing a specified number of substrings.

Syntax

Split(*Expression[***,** *Delimiter[***,** *Count[***,** *Compare]]]***)**

The **Split** function syntax has these parts:

Part	Description
Expression	Required. String expression containing substrings and delimiters. If *Expression* is a zero-length string, **Split** returns an empty array, that is, an array with no elements and no data.
Delimiter	Optional. String character used to identify substring limits. If omitted, the space character (" ") is assumed to be the delimiter. If *Delimiter* is a zero-length string, a single-element array containing the entire *Expression* string is returned.

Count	Optional. Number of substrings to be returned; -1 indicates that all substrings are returned.
Compare	Optional. Numeric value indicating the kind of comparison to use when evaluating substrings. See Settings section for values.

Settings

The *Compare* argument can have the following values:

Constant	Value	Description
vbBinaryCompare	0	Perform a binary comparison.
vbTextCompare	1	Perform a textual comparison.

Array Functions

Example

The following example uses the **Split** function to return an array from a string. The function performs a textual comparison of the delimiter, and returns all of the substrings.

```
Dim MyString, MyArray
MyString = Split("VBScriptXisXfun!", "x", -1, 1) '  MyString(0) contains "VBScript".
                                                 '  MyString(1) contains "is".
                                                 '  MyString(2) contains "fun!".
```

2.5.12 UBound Function

Returns the largest available subscript for the indicated dimension of an array.

Syntax

UBound(*ArrayName[***,** *Dimension]***)**

The **UBound** function syntax has these parts:

Part	Description
ArrayName	Required. Name of the array variable; follows standard variable naming conventions.
Dimension	Optional. Whole number indicating which dimension's upper bound is returned. Use 1 for the first dimension, 2 for the second, and so on. If *Dimension* is omitted, 1 is assumed.

Remarks

The **UBound** function is used with the LBound function to determine the size of an array. Use the LBound function to find the lower limit of an array dimension.

The lower bound for any dimension is always 0. As a result, **UBound** returns the following values for an array with these dimensions:

```
Dim A(100, 3, 4)
```

Statement	Return Value
UBound(A, 1**)**	100
UBound(A, 2**)**	3
UBound(A, 3**)**	4

Array Functions

2.6 String Functions

- **Asc** Function
- **Chr** Function
- **FormatCurrency** Function
- **FormatDateTime** Function
- **FormatNumber** Function
- **FormatPercent** Function
- **InStr** Function
- **InStrRev** Function
- **LCase** Function
- **Left** Function
- **Len** Function
- **LTrim** Function
- **MonthName** Function
- **Mid** Function
- **Replace** Function
- **Right** Function
- **RTrim** Function
- **Space** Function
- **StrComp** Function
- **String** Function
- **StrReverse** Function
- **Trim** Function
- **UCase** Function
- **WeekDayName** Function

2.6.1 Asc Function

Returns the ANSI character code corresponding to the first letter in a string.

Syntax

Asc(*String***)**

The *String* argument is any valid string expression. If the *String* contains no characters, a run-time error occurs.

NOTE

The **AscB** function is used with byte data contained in a string. Instead of returning the character code for the first character, **AscB** returns the first byte. **AscW** is provided for 32-bit platforms that use Unicode characters. It returns the Unicode (wide) character code, thereby avoiding the conversion from Unicode to ANSI.

String Functions

Conversion Functions

Example

In the following example, **Asc** returns the ANSI character code of the first letter of each string.

```
Dim MyNumber
MyNumber = Asc("A")      ' Returns 65.
MyNumber = Asc("a")      ' Returns 97.
MyNumber = Asc("Apple") ' Returns 65.
```

2.6.2 Chr Function

Returns the character associated with the specified ANSI character code.

Syntax

Chr(*CharCode***)**

The *CharCode* argument is a number that identifies a character.

Remarks

Numbers from 0 to 31 are the same as standard, nonprintable ASCII codes. For example, **Chr(**10**)** returns a linefeed character.

NOTE

The **ChrB** function is used with byte data contained in a string. Instead of returning a character, which may be one or two bytes, **ChrB** always returns a single byte. **ChrW** is provided for 32-bit platforms that use Unicode characters. Its argument is a Unicode (wide) character code, thereby avoiding the conversion from ANSI to Unicode.

String Functions

Conversion Functions

Example

The following example uses the **Chr** function to return the character associated with the specified character code.

```
Dim MyChar
MyChar = Chr(65) ' Returns A.
MyChar = Chr(97) ' Returns a.
MyChar = Chr(62) ' Returns >.
MyChar = Chr(37) ' Returns %.
```

2.6.3 FormatCurrency Function

Returns an expression formatted as a currency value using the currency symbol defined in the system control panel.

Syntax

FormatCurrency(*Expression[***,** *NumDigitsAfterDecimal[***,** *IncludeLeadingDigit[***,** *UseParensForNegativeNumbers[***,** *GroupDigits]]]]***)**

The **FormatCurrency** function syntax has these parts:

Part	Description
Expression	Required. Expression to be formatted.

NumDigitsAfterDecimal	Optional. Numeric value indicating how many places to the right of the decimal are displayed. Default value is -1, which indicates that the computer's regional settings are used.
IncludeLeadingDigit	Optional. Tristate constant that indicates whether or not a leading zero is displayed for fractional values. See Settings section for values.
UseParensForNegativeNumbers	Optional. Tristate constant that indicates whether or not to place negative values within parentheses. See Settings section for values.
GroupDigits	Optional. Tristate constant that indicates whether or not numbers are grouped using the group delimiter specified in the computer's regional settings. See Settings section for values.

Settings

The *IncludeLeadingDigit*, *UseParensForNegativeNumbers*, and *GroupDigits* arguments have the following settings:

Constant	Value	Description
vbTrue	-1	**True**
vbFalse	0	**False**
vbUseDefault	-2	Use the setting from the computer's regional settings.

Remarks

When one or more optional arguments are omitted, values for omitted arguments are provided by the computer's regional settings. The position of the currency symbol relative to the currency value is determined by the system's regional settings.

String Functions

Example

The following example uses the **FormatCurrency** function to format the expression as a currency and assign it to MyCurrency.

```
Dim MyCurrency
MyCurrency = FormatCurrency(1000) '  MyCurrency contains $1000.00.
```

2.6.4 FormatDateTime Function

Returns an expression formatted as a date or time.

Syntax

FormatDateTime(*Date[***, *NamedFormat]***)**

The **FormatDateTime** function syntax has these parts:

Part	Description
Date	Required. Date expression to be formatted.
NamedFormat	Optional. Numeric value that indicates the date/time format used. If omitted, **vbGeneralDate** is used.

Settings

The *NamedFormat* argument has the following settings:

Constant	Value	Description
vbGeneralDate	0	Display a date and/or time. If there is a date part, display it as a short date. If there is a time part, display it as a long time. If present, both parts are displayed.
vbLongDate	1	Display a date using the long date format specified in your computer's regional settings.

vbShortDate	2	Display a date using the short date format specified in your computer's regional settings.
vbLongTime	3	Display a time using the time format specified in your computer's regional settings.
vbShortTime	4	Display a time using the 24-hour format (hh:mm).

String Functions

Example

The following example uses the **FormatDateTime** function to format the expression as a long date and assign it to MyDateTime.

```
Function GetCurrentDate
  ' FormatDateTime formats Date in long date.
  GetCurrentDate = FormatDateTime( Date, 1)
End Function
```

2.6.5 FormatNumber Function

Returns an expression formatted as a number.

Syntax

FormatNumber(*Expression[***,** *NumDigitsAfterDecimal[***,** *IncludeLeadingDigit[***,** *UseParensForNegativeNumbers[***,** *GroupDigits]]]]***)**

The **FormatNumber** function syntax has these parts:

Part	Description
Expression	Required. Expression to be formatted.

NumDigitsAfterDecimal	Optional. Numeric value indicating how many places to the right of the decimal are displayed. Default value is -1, which indicates that the computer's regional settings are used.
IncludeLeadingDigit	Optional. Tristate constant that indicates whether or not a leading zero is displayed for fractional values. See Settings section for values.
UseParensForNegativeNumbers	Optional. Tristate constant that indicates whether or not to place negative values within parentheses. See Settings section for values.
GroupDigits	Optional. Tristate constant that indicates whether or not numbers are grouped using the group delimiter specified in the control panel. See Settings section for values.

Settings

The *IncludeLeadingDigit*, *UseParensForNegativeNumbers*, and *GroupDigits* arguments have the following settings:

Constant	Value	Description
vbTrue	-1	**True**
vbFalse	0	**False**
vbUseDefault	-2	Use the setting from the computer's regional settings.

Remarks

When one or more of the optional arguments are omitted, the values for omitted arguments are provided by the computer's regional settings.

String Functions

Example

The following example uses the **FormatNumber** function to format a number to have four decimal places.

```
Function FormatNumberDemo
  Dim MyAngle, MySecant, MyNumber
  MyAngle = 1.3                ' Define angle in radians.
  MySecant = 1 / Cos( MyAngle) ' Calculate secant.
  ' Format MySecant to four decimal places.
  FormatNumberDemo = FormatNumber( MySecant, 4)
End Function
```

2.6.6 FormatPercent Function

Returns an expression formatted as a percentage (multiplied by 100) with a trailing % character.

Syntax

FormatPercent(*Expression[***,** *NumDigitsAfterDecimal[***,** *IncludeLeadingDigit[***,** *UseParensForNegativeNumbers[***,** *GroupDigits]]]]***)**

The **FormatPercent** function syntax has these parts:

Part	Description
Expression	Required. Expression to be formatted.
NumDigitsAfterDecimal	Optional. Numeric value indicating how many places to the right of the decimal are displayed. Default value is -1, which indicates that the computer's regional settings are used.

IncludeLeadingDigit	Optional. Tristate constant that indicates whether or not a leading zero is displayed for fractional values. See Settings section for values.
UseParensForNegativeNumbers	Optional. Tristate constant that indicates whether or not to place negative values within parentheses. See Settings section for values.
GroupDigits	Optional. Tristate constant that indicates whether or not numbers are grouped using the group delimiter specified in the control panel. See Settings section for values.

Settings

The *IncludeLeadingDigit*, *UseParensForNegativeNumbers*, and *GroupDigits* arguments have the following settings:

Constant	Value	Description
vbTrue	-1	**True**
vbFalse	0	**False**
vbUseDefault	-2	Use the setting from the computer's regional settings.

Remarks

When one or more optional arguments are omitted, the values for the omitted arguments are provided by the computer's regional settings.

String Functions

Example

The following example uses the **FormatPercent** function to format an expression as a percent.

```
Dim MyPercent
MyPercent = FormatPercent( 2 / 32) '  MyPercent contains 6.25%.
```

2.6.7 InStr Function

Returns the position of the first occurrence of one string within another.

Syntax

InStr(*[Start***, ***]String1***, ***String2[***, ***Compare]***)**

The **InStr** function syntax has these parts:

Part	Description
Start	Optional. Numeric expression that sets the starting position for each search. If omitted, search begins at the first character position. If *Start* contains **Null**, an error occurs. The *Start* argument is required if *Compare* is specified.
String1	Required. String expression being searched.
String2	Required. String expression searched for.
Compare	Optional. Numeric value indicating the kind of comparison to use when evaluating substrings. See Settings section for values. If omitted, a binary comparison is performed.

Settings

The *Compare* argument can have the following values:

Constant	Value	Description
vbBinaryCompare	0	Perform a binary comparison.

vbTextCompare	1	Perform a textual comparison.

Return Values

The **InStr** function returns the following values:

If	InStr returns
String1 is zero-length	0
String1 is **Null**	**Null**
String2 is zero-length	*Start*
String2 is **Null**	**Null**
String2 is not found	0
String2 is found within *String1*	Position at which match is found.
Start > Len(*String2*)	0

NOTE

The **InStrB** function is used with byte data contained in a string. Instead of returning the character position of the first occurrence of one string within another, **InStrB** returns the byte position.

String Functions

Example

The following examples use **InStr** to search a string.

```
Dim SearchString, SearchChar, MyPos
SearchString ="XXpXXpXXPXXP" ' String to search in.
SearchChar = "P"              ' Search for "P".
' A textual comparison starting at position 4.
' Returns 6.
MyPos = Instr( 4, SearchString, SearchChar, 1)
' A binary comparison starting at position 1.
```

```
' Returns 9.
MyPos = Instr(1, SearchString, SearchChar, 0)
' Comparison is binary by default (last argument is omitted).
' Returns 9.
MyPos = Instr(SearchString, SearchChar)
' A binary comparison starting at position 1.
' Returns 0 ("W" is not found).
MyPos = Instr(1, SearchString, "W")
```

2.6.8 InStrRev Function

Returns the position of an occurrence of one string within another, from the end of string.

Syntax

InStrRev(*String1***, *String2*[, *Start*[, *Compare*]])**

The **InStrRev** function syntax has these parts:

Part	Description
String1	Required. String expression being searched.
String2	Required. String expression being searched for.
Start	Optional. Numeric expression that sets the starting position for each search. If omitted, -1 is used, which means that the search begins at the last character position. If *Start* contains **Null**, an error occurs.
Compare	Optional. Numeric value indicating the kind of comparison to use when evaluating substrings. If omitted, a binary comparison is performed. See Settings section for values.

Settings

The *Compare* argument can have the following values:

Constant	Value	Description
vbBinaryCompare	0	Perform a binary comparison.

vbTextCompare	1	Perform a textual comparison.

Return Values

InStrRev returns the following values:

If	InStrRev returns
String1 is zero-length	0
String1 is **Null**	**Null**
String2 is zero-length	*Start*
String2 is **Null**	**Null**
String2 is not found	0
String2 is found within *String1*	Position at which match is found.
Start > Len(*String2*)	0

NOTE

The syntax for the **InStrRev** function is not the same as the syntax for the InStr function.

String Functions

Example

The following examples use the **InStrRev** function to search a string.

```
Dim SearchString, SearchChar, MyPos
SearchString ="XXpXXpXXPXXP" ' String to search in.
SearchChar = "P"              ' Search for "P".
' A binary comparison starting at position 10.
' Returns 9.
MyPos = InstrRev( SearchString, SearchChar, 10, 0)
' A textual comparison starting at the last position.
' Returns 12.
MyPos = InstrRev( SearchString, SearchChar, -1, 1)
' Comparison is binary by default (last argument is omitted).
' Returns 0.
```

```
MyPos = InstrRev( SearchString, SearchChar, 8)
```

2.6.9 LCase Function

Returns a string that has been converted to lowercase.

Syntax

LCase(*String***)**

The *String* argument is any valid string expression. If *String* contains **Null**, **Null** is returned.

Remarks

Only uppercase letters are converted to lowercase; all lowercase letters and nonletter characters remain unchanged.

String Functions

Conversion Functions

Example

The following example uses the **LCase** function to convert uppercase letters to lowercase.

```
Dim MyString
Dim LCaseString
MyString = "VBSCript"
LCaseString = LCase( MyString) ' LCaseString contains "vbscript".
```

2.6.10 Left Function

Returns a specified number of characters from the left side of a string.

Syntax

Left(*String***, *Length*)**

The **Left** function syntax has these parts:

Part	Description

String	String expression from which the leftmost characters are returned. If *String* contains **Null**, **Null** is returned.
Length	Numeric expression indicating how many characters to return. If 0, a zero-length string("") is returned. If greater than or equal to the number of characters in *String*, the entire string is returned.

Remarks

To determine the number of characters in *String*, use the Len function.

NOTE

The **LeftB** function is used with byte data contained in a string. Instead of specifying the number of characters to return, *Length* specifies the number of bytes.

String Functions

Example

The following example uses the **Left** function to return the first three characters of MyString.

```
Dim MyString, LeftString
MyString = "VBSCript"
LeftString = Left( MyString, 3) '  LeftString contains "VBS".
```

2.6.11 Len Function

Returns the number of characters in a string or the number of bytes required to store a variable.

Syntax

Len(*String* | *VarName***)**

The **Len** function syntax has these parts:

Part	Description

String	Any valid string expression. If *String* contains **Null**, **Null** is returned.
VarName	Any valid variable name. If *VarName* contains **Null**, **Null** is returned.

NOTE

The **LenB** function is used with byte data contained in a string. Instead of returning the number of characters in a string, **LenB** returns the number of bytes used to represent that string.

String Functions

Example

The following example uses the **Len** function to return the number of characters in a string.

```
Dim MyString
MyString = Len("VBSCRIPT") '  MyString contains 8.
```

2.6.12 LTrim Function

Returns a copy of a string without leading spaces (RTrim without trailing spaces, Trim without both leading and trailing spaces).

Syntax

LTrim(*String***)**

The *String* argument is any valid string expression. If *String* contains **Null**, **Null** is returned.

String Functions

Example

The following example uses the **LTrim**, RTrim, and Trim functions to trim leading spaces, trailing spaces, and both leading and trailing spaces, respectively.

```
Dim MyVar
MyVar = LTrim("   vbscript ") ' MyVar contains "vbscript ".
MyVar = RTrim("   vbscript ") ' MyVar contains "   vbscript".
MyVar = Trim("   vbscript ")  ' MyVar contains "vbscript".
```

2.6.13 Mid Function

Returns a specified number of characters from a string.

Syntax

Mid(*String***, *Start*[, *Length*])**

The **Mid** function syntax has these parts:

Part	Description
String	String expression from which characters are returned. If *String* contains **Null**, **Null** is returned.
Start	Character position in *String* at which the part to be taken begins. If *Start* is greater than the number of characters in *String*, **Mid** returns a zero-length string ("").
Length	Number of characters to return. If omitted or if there are fewer than *Length* characters in the text (including the character at *Start*), all characters from the *Start* position to the end of the string are returned.

Remarks

To determine the number of characters in *String*, use the Len function.

NOTE

The **MidB** function is used with byte data contained in a string. Instead of specifying the number of characters, the arguments specify numbers of bytes.

String Functions

Example

The following example uses the **Mid** function to return six characters, beginning with the fourth character, in a string.

```
Dim MyVar
MyVar = Mid("VB Script is fun!", 4, 6) ' MyVar contains "Script".
```

2.6.14 MonthName Function

Returns a string indicating the specified month.

Syntax

MonthName(*Month[***,** *Abbreviate]***)**

The **MonthName** function syntax has these parts:

Part	Description
Month	Required. The numeric designation of the month. For example, January is 1, February is 2, and so on.
Abbreviate	Optional. Boolean value that indicates if the month name is to be abbreviated. If omitted, the default is **False**, which means that the month name is not abbreviated.

Date and Time Functions

String Functions

Example

The following example uses the **MonthName** function to return an abbreviated month name for a date expression.

```
Dim MyVar
MyVar = MonthName(10, True) '  MyVar contains "Oct".
```

2.6.15 Replace Function

Returns a string in which a specified substring has been replaced with another substring a specified number of times.

Syntax

Replace(*Expression*, *Find*, *ReplaceWith*[, *Start*[, *Count*[, *Compare*]]]**)**

The **Replace** function syntax has these parts:

Part	Description
Expression	Required. String expression containing substring to replace.
Find	Required. Substring being searched for.
ReplaceWith	Required. Replacement substring.
Start	Optional. Position within *Expression* where substring search is to begin. If omitted, 1 is assumed. Must be used in conjunction with *Count*.
Count	Optional. Number of substring substitutions to perform. If omitted, the default value is -1, which means make all possible substitutions. Must be used in conjunction with *Start*.
Compare	Optional. Numeric value indicating the kind of comparison to use when evaluating substrings. See Settings section for values. If omitted, the default value is 0, which means perform a binary comparison.

Settings

The *Compare* argument can have the following values:

Constant	Value	Description
vbBinaryCompare	0	Perform a binary comparison.
vbTextCompare	1	Perform a textual comparison.

Return Values

Replace returns the following values:

If	Replace returns
Expression is zero-length	Zero-length string ("").
Expression is **Null**	An error.
Find is zero-length	Copy of *Expression.*
ReplaceWith is zero-length	Copy of *Expression* with all occurences of *Find* removed.
Start > Len(*Expression*)	Zero-length string.
Count is 0	Copy of *Expression.*

Remarks

The return value of the **Replace** function is a string, with substitutions made, that begins at the position specified by *Start* and and concludes at the end of the *Expression* string. It is not a copy of the original string from start to finish.

String Functions

Example

The following example uses the **Replace** function to return a string.

```
Dim MyString
```

```
' A binary comparison starting at the beginning of the string.
' Returns "XXYXXPXXY".
MyString = Replace("XXpXXPXXp", "p", "Y")
' A textual comparison starting at position 3.
' Returns "YXXYXXY".
MyString = Replace("XXpXXPXXp", "p", "Y", 3, -1, 1)
```

2.6.16 Right Function

Returns a specified number of characters from the right side of a string.

Syntax

Right(*String*, *Length***)**

The **Right** function syntax has these parts:

Part	Description
String	String expression from which the rightmost characters are returned. If *String* contains **Null**, **Null** is returned.
Length	Numeric expression indicating how many characters to return. If 0, a zero-length string is returned. If greater than or equal to the number of characters in *String*, the entire string is returned.

Remarks

To determine the number of characters in *String*, use the Len function.

NOTE

The **RightB** function is used with byte data contained in a string. Instead of specifying the number of characters to return, **Length** specifies the number of bytes.

String Functions

Example

The following example uses the **Right** function to return a specified number of characters from the right side of a string.

```
Dim AnyString, MyStr
AnyString = "Hello World"    ' Define string.
MyStr = Right( AnyString, 1)  ' Returns "d".
MyStr = Right( AnyString, 6)  ' Returns " World".
MyStr = Right( AnyString, 20) ' Returns "Hello World".
```

2.6.17 RTrim Function

Returns a copy of a string without trailing spaces (LTrim without leading spaces, Trim without both leading and trailing spaces).

Syntax

RTrim(*String***)**

The *String* argument is any valid string expression. If *String* contains **Null**, **Null** is returned.

 String Functions

Example

The following example uses the LTrim, **RTrim**, and Trim functions to trim leading spaces, trailing spaces, and both leading and trailing spaces, respectively.

```
Dim MyVar
MyVar = LTrim("   vbscript ") ' MyVar contains "vbscript ".
MyVar = RTrim("   vbscript ") ' MyVar contains "   vbscript".
MyVar = Trim("   vbscript ")  ' MyVar contains "vbscript".
```

2.6.18 Space Function

Returns a string consisting of the specified number of spaces.

Syntax

Space(*Number***)**

The *Number* argument is the number of spaces you want in the string.

String Functions

Example

The following example uses the **Space** function to return a string consisting of a specified number of spaces.

```
Dim MyString
' Returns a string with 10 spaces.
MyString = Space(10)
' Insert 10 spaces between two strings.
MyString = "Hello" & Space(10) & "World"
```

2.6.19 StrComp Function

Returns a value indicating the result of a string comparison.

Syntax

StrComp(*String1***, *String2[*, *Compare]***)**

The **StrComp** function syntax has these parts:

Part	Description
String1	Required. Any valid string expression.
String2	Required. Any valid string expression.
Compare	Optional. Numeric value indicating the kind of comparison to use when evaluating strings. If omitted, a binary comparison is performed. See Settings section for values.

Settings

The *Compare* argument can have the following values:

Constant	Value	Description
vbBinaryCompare	0	Perform a binary comparison.
vbTextCompare	1	Perform a textual comparison.

Return Values

The **StrComp** function has the following return values:

If	StrComp returns
String1 is less than *String2*	-1
String1 is equal to *String2*	0
String1 is greater than *String2*	1
String1 or *String2* is **Null**	**Null**

String Functions

Example

The following example uses the **StrComp** function to return the results of a string comparison. If the third argument is 1, a textual comparison is performed; if the third argument is 0 or omitted, a binary comparison is performed.

```
Dim MyStr1, MyStr2, MyComp
MyStr1 = "ABCD"
MyStr2 = "abcd"                        ' Define variables.
MyComp = StrComp( MyStr1, MyStr2, 1) ' Returns 0.
MyComp = StrComp( MyStr1, MyStr2, 0) ' Returns -1.
MyComp = StrComp( MyStr2, MyStr1)     ' Returns 1.
```

2.6.20 String Function

Returns a repeating character string of the length specified.

Syntax

String(*Number*, *Character*)

The **String** function syntax has these parts:

Part	Description
Number	Length of the returned string. If *Number* contains **Null**, **Null** is returned.
Character	Character code specifying the character or string expression whose first character is used to build the return string. If *Character* contains **Null**, **Null** is returned.

Remarks

If you specify a number for *Character* greater than 255, **String** converts the number to a valid character code using the formula:

character `Mod 256`

String Functions

Example

The following example uses the **String** function to return repeating character strings of the length specified.

```
Dim MyString
MyString = String( 5, "*")     ' Returns "*****".
MyString = String( 5, 42)      ' Returns "*****".
MyString = String( 10, "ABC") ' Returns "AAAAAAAAAA".
```

2.6.21 StrReverse Function

Returns a string in which the character order of a specified string is reversed.

Syntax

StrReverse(*String***)**

The *String* argument is the string whose characters are to be reversed. If *String* is a zero-length string (""), a zero-length string is returned. If *String* is **Null**, an error occurs.

String Functions

Example

The following example uses the **StrReverse** function to return a string in reverse order.

```
Dim MyStr
MyStr = StrReverse("VBScript") ' MyStr contains "tpircSBV".
```

2.6.22 Trim Function

Returns a copy of a string without both leading and trailing spaces (LTrim without leading spaces, RTrim without trailing spaces).

Syntax

Trim(*String***)**

The *String* argument is any valid string expression. If *String* contains **Null**, **Null** is returned.

String Functions

Example

The following example uses the LTrim, RTrim, and **Trim** functions to trim leading spaces, trailing spaces, and both leading and trailing spaces, respectively.

```
Dim MyVar
MyVar = LTrim("   vbscript ") ' MyVar contains "vbscript ".
MyVar = RTrim("   vbscript ") ' MyVar contains "   vbscript".
MyVar = Trim("   vbscript ")  ' MyVar contains "vbscript".
```

2.6.23 UCase Function

Returns a string that has been converted to uppercase.

Syntax

UCase(*String***)**

The *String* argument is any valid string expression. If *String* contains **Null**, **Null** is returned.

Remarks

Only lowercase letters are converted to uppercase; all uppercase letters and nonletter characters remain unchanged.

String Functions

Conversion Functions

Example

The following example uses the **UCase** function to return an uppercase version of a string.

```
Dim MyWord
MyWord = UCase("Hello World") '  Returns "HELLO WORLD".
```

2.6.24 WeekdayName Function

Returns a string indicating the specified day of the week.

Syntax

WeekdayName(*WeekDay[, Abbreviate[***, ***FirstDayOfWeek]]***)**

The **WeekdayName** function syntax has these parts:

Part	Description

WeekDay	Required. The numeric designation for the day of the week. Numeric value of each day depends on setting of the *FirstDayOfWeek* setting.
Abbreviate	Optional. Boolean value that indicates if the weekday name is to be abbreviated. If omitted, the default is **False**, which means that the weekday name is not abbreviated.
FirstDayOfWeek	Optional. Numeric value indicating the first day of the week. See Settings section for values.

Settings

The *FirstDayOfWeek* argument can have the following values:

Constant	Value	Description
vbUseSystem	0	Use National Language Support (NLS) API setting.
vbSunday	1	Sunday
vbMonday	2	Monday
vbTuesday	3	Tuesday
vbWednesday	4	Wednesday
vbThursday	5	Thursday
vbFriday	6	Friday
vbSaturday	7	Saturday

Date and Time Functions

String Functions

Example

The following example uses the **WeekDayName** function to return the specified day.

```
Dim MyDate
MyDate = WeekDayName( 6, True) '  MyDate contains Fri.
```

2.7 Conversion Functions

- **Asc** Function
- **CBool** Function
- **CByte** Function
- **CCur** Function
- **CDate** Function
- **CDbl** Function
- **Chr** Function
- **CInt** Function
- **CLng** Function
- **CSng** Function
- **CStr** Function
- **DateSerial** Function
- **DateValue** Function
- **Day** Function
- **Fix** Function
- **Hex** Function
- **Hour** Function
- **Int** Function
- **LCase** Function
- **Minute** Function
- **Month** Function
- **Oct** Function
- **Second** Function
- **TimeSerial** Function
- **TimeValue** Function
- **UCase** Function
- **WeekDay** Function

Year Function

2.7.1 Asc Function

Returns the ANSI character code corresponding to the first letter in a string.

Syntax

Asc(*String***)**

The *String* argument is any valid string expression. If the *String* contains no characters, a run-time error occurs.

NOTE

The **AscB** function is used with byte data contained in a string. Instead of returning the character code for the first character, **AscB** returns the first byte. **AscW** is provided for 32-bit platforms that use Unicode characters. It returns the Unicode (wide) character code, thereby avoiding the conversion from Unicode to ANSI.

String Functions

Conversion Functions

Example

In the following example, **Asc** returns the ANSI character code of the first letter of each string.

```
Dim MyNumber
MyNumber = Asc("A")      ' Returns 65.
MyNumber = Asc("a")      ' Returns 97.
MyNumber = Asc("Apple") ' Returns 65.
```

2.7.2 CBool Function

Returns an expression that has been converted to a **Variant** of subtype **Boolean**.

Syntax

CBool(*Expression***)**

The *Expression* argument is any valid expression.

Remarks

If *Expression* is zero, **False** is returned; otherwise, **True** is returned. If *Expression* can't be interpreted as a numeric value, a run-time error occurs.

Data Type Functions

Conversion Functions

Example

The following example uses the **CBool** function to convert an expression to a **Boolean**. If the expression evaluates to a nonzero value, **CBool** returns **True**; otherwise, it returns **False**.

```
Dim A, B, Check
A = 5
B = 5                ' Initialize variables.
Check = CBool( A = B) ' Check contains True.
A = 0                ' Define variable.
Check = CBool( A)     ' Check contains False.
```

2.7.3 CByte Function

Returns an expression that has been converted to a **Variant** of subtype **Byte**.

Syntax

CByte(*Expression***)**

The *Expression* argument is any valid expression.

Remarks

In general, you can document your code using the subtype conversion functions to show that the result of some operation should be expressed as a particular data type rather than the default data type. For example, use **CByte** to force byte arithmetic in cases where currency, single-precision, double-precision, or integer arithmetic normally would occur.

Use the **CByte** function to provide internationally aware conversions from any other data type to a **Byte** subtype. For example, different decimal separators are properly recognized depending on the locale setting of your system, as are different thousand separators.

If *Expression* lies outside the acceptable range for the **Byte** subtype, an error occurs.

Data Type Functions

Conversion Functions

Example

The following example uses the **CByte** function to convert an expression to a byte.

```
Dim MyDouble, MyByte
MyDouble = 125.5678      ' MyDouble is a Double.
MyByte = CByte(MyDouble) ' MyByte contains 126.
```

2.7.4 CCur Function

Returns an expression that has been converted to a **Variant** of subtype **Currency**.

Syntax

CCur(*Expression***)**

The *Expression* argument is any valid expression.

Remarks

In general, you can document your code using the subtype conversion functions to show that the result of some operation should be expressed as a particular data type rather than the default data type. For example, use **CCur** to force currency arithmetic in cases where integer arithmetic normally would occur.

You should use the **CCur** function to provide internationally aware conversions from any other data type to a **Currency** subtype. For example, different decimal separators and thousands separators are properly recognized depending on the locale setting of your system.

Data Type Functions

Conversion Functions

Example

The following example uses the **CCur** function to convert an expression to a Currency.

```
Dim MyDouble, MyCurr
MyDouble = 543.214588 ' MyDouble is a Double.
' Convert result of MyDouble * 2 (1086.429176) to a Currency (1086.4292).
MyCurr = CCur(MyDouble * 2)
```

2.7.5 CDate Function

Returns an expression that has been converted to a **Variant** of subtype **Date**.

Syntax

CDate(*Date***)**

The *Date* argument is any valid date expression.

Remarks

Use the IsDate function to determine if *Date* can be converted to a date or time. **CDate** recognizes date literals and time literals as well as some numbers that fall within the range of acceptable dates. When converting a number to a date, the whole number portion is converted to a date. Any fractional part of the number is converted to a time of day, starting at midnight.

CDate recognizes date formats according to the locale setting of your system. The correct order of day, month, and year may not be determined if it is provided in a format other than one of the recognized date settings. In addition, a long date format is not recognized if it also contains the day-of-the-week string.

Data Type Functions

Conversion Functions

Example

The following example uses the **CDate** function to convert a string to a date. In general, hard coding dates and times as strings (as shown in this example) is not recommended. Use date and time literals (such as `#10/19/1962#`, `#4:45:23 PM#`) instead.

```
Dim MyShortTime, MyDate, MyTime
MyDate = "October 19, 1962" ' Define date.
MyShortDate = CDate(MyDate) ' Convert to Date data type.
MyTime = "4:35:47 PM"       ' Define time.
MyShortTime = CDate(MyTime) ' Convert to Date data type.
```

2.7.6 CDbl Function

Returns an expression that has been converted to a **Variant** of subtype **Double**.

Syntax

CDbl(*Expression***)**

The *Expression* argument is any valid expression.

Remarks

In general, you can document your code using the subtype conversion functions to show that the result of some operation should be expressed as a particular data type rather than the default data type. For example, use **CDbl** or CSng to force double-precision or single-precision arithmetic in cases where currency or integer arithmetic normally would occur.

Use the **CDbl** function to provide internationally aware conversions from any other data type to a **Double** subtype. For example, different decimal separators and thousands separators are properly recognized depending on the locale setting of your system.

Data Type Functions

Conversion Functions

Example

This example uses the **CDbl** function to convert an expression to a **Double**.

```
Dim MyCurr, MyDouble
MyCurr = CCur(234.456784) ' MyCurr is a Currency (234.4567).
' Convert result to a Double (19.2254576).
MyDouble = CDbl(MyCurr * 8.2 * 0.01)
```

2.7.7 Chr Function

Returns the character associated with the specified ANSI character code.

Syntax

Chr(*CharCode***)**

The *CharCode* argument is a number that identifies a character.

Remarks

Numbers from 0 to 31 are the same as standard, nonprintable ASCII codes. For example, **Chr(**10**)** returns a linefeed character.

NOTE

The **ChrB** function is used with byte data contained in a string. Instead of returning a character, which may be one or two bytes, **ChrB** always returns a single byte. **ChrW** is provided for 32-bit platforms that use Unicode characters. Its argument is a Unicode (wide) character code, thereby avoiding the conversion from ANSI to Unicode.

String Functions

Conversion Functions

Example

The following example uses the **Chr** function to return the character associated with the specified character code.

```
Dim MyChar
MyChar = Chr(65) ' Returns A.
MyChar = Chr(97) ' Returns a.
MyChar = Chr(62) ' Returns >.
MyChar = Chr(37) ' Returns %.
```

2.7.8 CInt Function

Returns an expression that has been converted to a **Variant** of subtype **Integer**.

Syntax

CInt(*Expression***)**

The *Expression* argument is any valid expression.

Remarks

In general, you can document your code using the subtype conversion functions to show that the result of some operation should be expressed as a particular data type rather than the default data type. For example, use **CInt** or CLng to force integer arithmetic in cases where currency, single-precision, or double-precision arithmetic normally would occur.

Use the **CInt** function to provide internationally aware conversions from any other data type to an **Integer** subtype. For example, different decimal separators are properly recognized depending on the locale setting of your system, as are different thousand separators.

If *Expression* lies outside the acceptable range for the **Integer** subtype, an error occurs.

NOTE

CInt differs from the Fix and Int functions, which truncate, rather than round, the fractional part of a number. When the fractional part is exactly 0.5, the **CInt** function always rounds it to the nearest even number. For example, 0.5 rounds to 0, and 1.5 rounds to 2.

Data Type Functions

Conversion Functions

Example

The following example uses the **CInt** function to convert a value to an Integer.

```
Dim MyDouble, MyInt
MyDouble = 2345.5678   ' MyDouble is a Double.
MyInt = CInt(MyDouble) ' MyInt contains 2346.
```

2.7.9 CLng Function

Returns an expression that has been converted to a **Variant** of subtype **Long**.

Syntax

CLng(*Expression***)**

The *Expression* argument is any valid expression.

Remarks

In general, you can document your code using the subtype conversion functions to show that the result of some operation should be expressed as a particular data type rather than the default data type. For example, use CInt or **CLng** to force integer arithmetic in cases where currency, single-precision, or double-precision arithmetic normally would occur.

Use the **CLng** function to provide internationally aware conversions from any other data type to a **Long** subtype. For example, different decimal separators are properly recognized depending on the locale setting of your system, as are different thousand separators.

If *Expression* lies outside the acceptable range for the **Long** subtype, an error occurs.

NOTE

CLng differs from the Fix and Int functions, which truncate, rather than round, the fractional part of a number. When the fractional part is exactly 0.5, the **CLng** function always rounds it to the nearest even number. For example, 0.5 rounds to 0, and 1.5 rounds to 2.

Data Type Functions

Conversion Functions

Example

The following example uses the **CLng** function to convert a value to a **Long**.

```
Dim MyVal1, MyVal2, MyLong1, MyLong2
MyVal1 = 25427.45
MyVal2 = 25427.55       ' MyVal1, MyVal2 are Doubles.
MyLong1 = CLng(MyVal1) ' MyLong1 contains 25427.
```

```
MyLong2 = CLng( MyVal2) ' MyLong2 contains 25428.
```

2.7.10 CSng Function

Returns an expression that has been converted to a **Variant** of subtype **Single**.

Syntax

CSng(*Expression***)**

The *Expression* argument is any valid expression.

Remarks

In general, you can document your code using the data type conversion functions to show that the result of some operation should be expressed as a particular data type rather than the default data type. For example, use CDbl or **CSng** to force double-precision or single-precision arithmetic in cases where currency or integer arithmetic normally would occur.

Use the **CSng** function to provide internationally aware conversions from any other data type to a **Single** subtype. For example, different decimal separators are properly recognized depending on the locale setting of your system, as are different thousand separators.

If *Expression* lies outside the acceptable range for the **Single** subtype, an error occurs.

Data Type Functions

Conversion Functions

Example

The following example uses the **CSng** function to convert a value to a **Single**.

```
Dim MyDouble1, MyDouble2    ' MyDouble1, MyDouble2 are Doubles.
Dim MySingle1, MySingle2
MyDouble1 = 75.3421115
MyDouble2 = 75.3421555
MySingle1 = CSng( MyDouble1) ' MySingle1 contains 75.34211.
MySingle2 = CSng( MyDouble2) ' MySingle2 contains 75.34216.
```

2.7.11 CStr Function

Returns an expression that has been converted to a **Variant** of subtype **String**.

Syntax

CStr(*Expression***)**

The *Expression* argument is any valid expression.

Remarks

In general, you can document your code using the data type conversion functions to show that the result of some operation should be expressed as a particular data type rather than the default data type. For example, use **CStr** to force the result to be expressed as a **String**.

You should use the **CStr** function instead of String to provide internationally aware conversions from any other data type to a **String** subtype. For example, different decimal separators are properly recognized depending on the locale setting of your system.

The data in *Expression* determines what is returned according to the following table.

If *Expression* **is**	**CStr returns**
Boolean	A **String** containing **True** or **False**.
Date	A **String** containing a date in the short-date format of your system.
Null	A run-time error.
Empty	A zero-length **String** ("").
Error	A **String** containing the word Error followed by the error number.
Other numeric	A **String** containing the number.

Data Type Functions

Conversion Functions

Example

The following example uses the **CStr** function to convert a numeric value to a **String**.

```
Dim MyDouble, MyString
MyDouble = 437.324        ' MyDouble is a Double.
MyString = CStr(MyDouble) ' MyString contains "437.324".
```

2.7.12 DateSerial Function

Returns a **Variant** of subtype **Date** for a specified year, month, and day.

Syntax

DateSerial(*Year***,** *Month***,** *Day***)**

The **DateSerial** function syntax has these parts:

Part	Description
Year	Number between 100 and 9999, inclusive, or a numeric expression.
Month	Any numeric expression.
Day	Any numeric expression.

Remarks

To specify a date, such as December 31, 1991, the range of numbers for each **DateSerial** argument should be in the accepted range for the unit; that is, 1–31 for days and 1–12 for months. However, you can also specify relative dates for each argument using any numeric expression that represents some number of days, months, or years before or after a certain date.

For the *Year* argument, values between 0 and 99, inclusive, are interpreted as the years 1900–1999. For all other *Year* arguments, use a complete four-digit year (for example, 1800).

When any argument exceeds the accepted range for that argument, it increments to the next larger unit as appropriate. For example, if you specify 35 days, it is evaluated as one month and some number of days, depending on where in the year it is applied. However, if any single argument is outside the range -32,768 to 32,767, or if the date specified by the three arguments, either directly or by expression, falls outside the acceptable range of dates, an error occurs.

Date and Time Functions

Conversion Functions

Example

The following example uses numeric expressions instead of absolute date numbers. Here the **DateSerial** function returns a date that is the day before the first day (1 - 1) of two months before August (8 - 2) of 10 years before 1990 (1990 - 10); in other words, May 31, 1980.

```
Dim MyDate1, MyDate2
MyDate1 = DateSerial(1970, 1, 1)              ' Returns January 1, 1970.
MyDate2 = DateSerial(1990 - 10, 8 - 2, 1 - 1) ' Returns May 31, 1980.
```

2.7.13 DateValue Function

Returns a **Variant** of subtype **Date**.

Syntax

DateValue(*Date***)**

The *Date* argument is normally a string expression representing a date from January 1, 100 through December 31, 9999. However, *Date* can also be any expression that can represent a date, a time, or both a date and time, in that range.

Remarks

If the *Date* argument includes time information, **DateValue** doesn't return it. However, if *Date* includes invalid time information (such as "89:98"), an error occurs.

If *Date* is a string that includes only numbers separated by valid date separators, **DateValue** recognizes the order for month, day, and year according to the short date format you specified for your system. **DateValue** also recognizes unambiguous dates that contain month names, either in long or abbreviated form. For example, in addition to recognizing 12/30/1991 and 12/30/91, **DateValue** also recognizes December 30, 1991 and Dec 30, 1991.

If the year part of *Date* is omitted, **DateValue** uses the current year from your computer's system date.

Date and Time Functions

Conversion Functions

Example

The following example uses the **DateValue** function to convert a string to a date. You can also use date literals to directly assign a date to a **Variant** variable, for example, MyDate = #9/11/63#.

```
Dim MyDate
MyDate = DateValue("September 11, 1963") ' Return a date.
```

2.7.14 Day Function

Returns a whole number between 1 and 31, inclusive, representing the day of the month.

Syntax

Day(*Date***)**

The *Date* argument is any expression that can represent a date. If *Date* contains **Null**, **Null** is returned.

Date and Time Functions

Conversion Functions

Example

The following example uses the **Day** function to obtain the day of the month from a specified date.

```
Dim MyDay
MyDay = Day("October 19, 1962") ' MyDay contains 19.
```

2.7.15 Fix Function

Returns the integer portion of a number.

Syntax

Fix(*Number***)**

The *Number* argument can be any valid numeric expression. If *Number* contains **Null**, **Null** is returned.

Remarks

Both Int and **Fix** remove the fractional part of *Number* and return the resulting integer value.

The difference between Int and **Fix** is that if *Number* is negative, Int returns the first negative integer less than or equal to *Number,* whereas **Fix** returns the first negative integer greater than or equal to *Number.* For example, Int converts -8.4 to -9, and **Fix** converts -8.4 to -8.

Fix(*Number*) is equivalent to:

```
Sgn( Number) * Int( Abs( Number))
```

- Data Type Functions
- Conversion Functions
- Math Functions

Example

The following examples illustrate how the Int and **Fix** functions return integer portions of numbers.

```
Dim MyNumber
MyNumber = Int( 99.8)   ' Returns 99.
MyNumber = Fix( 99.2)   ' Returns 99.
MyNumber = Int( -99.8) ' Returns -100.
MyNumber = Fix( -99.8) ' Returns -99.
```

```
MyNumber = Int(-99.2) ' Returns -100.
MyNumber = Fix(-99.2) ' Returns -99.
```

2.7.16 Hex Function

Returns a string representing the hexadecimal value of a number.

Syntax

Hex(*Number***)**

The *Number* argument is any valid expression.

Remarks

If *Number* is not already a whole number, it is rounded to the nearest whole number before being evaluated.

If **Number** *is*	**Hex returns**
Null	**Null**
Empty	Zero (0).
Any other number	Up to eight hexadecimal characters.

You can represent hexadecimal numbers directly by preceding numbers in the proper range with &H. For example, &H10 represents decimal 16 in hexadecimal notation.

Conversion Functions

Example

The following example uses the **Hex** function to return the hexadecimal value of a number.

```
Dim MyHex
MyHex = Hex(5)    ' Returns 5.
MyHex = Hex(10)   ' Returns A.
MyHex = Hex(459)  ' Returns 1CB.
```

2.7.17 Hour Function

Returns a whole number between 0 and 23, inclusive, representing the hour of the day.

Syntax

Hour(*Time***)**

The *Time* argument is any expression that can represent a time. If *Time* contains **Null**, **Null** is returned.

Date and Time Functions

Conversion Functions

Example

The following example uses the **Hour** function to obtain the hour from the current time.

```
Dim MyTime, MyHour
MyTime = Now
' MyHour contains the number representing the current hour.
MyHour = Hour( MyTime)
```

2.7.18 Int Function

Returns the integer portion of a number.

Syntax

Int(*Number***)**

The *Number* argument can be any valid numeric expression. If *Number* contains **Null**, **Null** is returned.

Remarks

Both **Int** and Fix remove the fractional part of *Number* and return the resulting integer value.

The difference between **Int** and Fix is that if *Number* is negative, **Int** returns the first negative integer less than or equal to *Number,* whereas Fix returns the first negative integer greater than or equal to *Number.* For example, **Int** converts -8.4 to -9, and Fix converts -8.4 to -8.

Fix(*Number*) is equivalent to.

```
Sgn( Number) * Int( Abs( Number))
```

- Data Type Functions
- Conversion Functions
- Math Functions

Example

The following examples illustrate how the **Int** and Fix functions return integer portions of numbers.

```
Dim MyNumber
MyNumber = Int( 99.8)   ' Returns 99.
MyNumber = Fix( 99.2)   ' Returns 99.
MyNumber = Int( -99.8) ' Returns -100.
MyNumber = Fix( -99.8) ' Returns -99.
MyNumber = Int( -99.2) ' Returns -100.
MyNumber = Fix( -99.2) ' Returns -99.
```

2.7.19 LCase Function

Returns a string that has been converted to lowercase.

Syntax

LCase(*String***)**

The *String* argument is any valid string expression. If *String* contains **Null**, **Null** is returned.

Remarks

Only uppercase letters are converted to lowercase; all lowercase letters and nonletter characters remain unchanged.

String Functions

Conversion Functions

Example

The following example uses the **LCase** function to convert uppercase letters to lowercase.

```
Dim MyString
Dim LCaseString
MyString = "VBSCript"
LCaseString = LCase( MyString) '  LCaseString contains "vbscript".
```

2.7.20 Minute Function

Returns a whole number between 0 and 59, inclusive, representing the minute of the hour.

Syntax

Minute(*Time***)**

The *Time* argument is any expression that can represent a time. If *Time* contains **Null**, **Null** is returned.

Date and Time Functions

Conversion Functions

Example

The following example uses the **Minute** function to return the minute of the hour.

```
Dim MyVar
MyVar = Minute( Now)
```

2.7.21 Month Function

Returns a whole number between 1 and 12, inclusive, representing the month of the year.

Syntax

Month(*Date***)**

The *Date* argument is any expression that can represent a date. If *Date* contains **Null**, **Null** is returned.

Date and Time Functions

Conversion Functions

Example

The following example uses the **Month** function to return the current month.

```
Dim MyVar
' MyVar contains the number corresponding to the current month.
MyVar = Month( Now)
```

2.7.22 Oct Function

Returns a string representing the octal value of a number.

Syntax

Oct(Number)

The *Number* argument is any valid expression.

Remarks

If *Number* is not already a whole number, it is rounded to the nearest whole number before being evaluated.

If **Number** *is*	**Oct returns**
Null	**Null**
Empty	Zero (0).
Any other number	Up to 11 octal characters.

You can represent octal numbers directly by preceding numbers in the proper range with &O. For example, &O10 is the octal notation for decimal 8.

Conversion Functions

Example

The following example uses the **Oct** function to return the octal value of a number.

```
Dim MyOct
MyOct = Oct( 4)    ' Returns 4.
MyOct = Oct( 8)    ' Returns 10.
MyOct = Oct( 459) ' Returns 713.
```

2.7.23 Second Function

Returns a whole number between 0 and 59, inclusive, representing the second of the minute.

Syntax

Second(*Time***)**

The *Time* argument is any expression that can represent a time. If *Time* contains **Null**, **Null** is returned.

Date and Time Functions

Conversion Functions

Example

The following example uses the **Second** function to return the current second.

```
Dim MySec
' MySec contains the number representing the current second.
MySec = Second( Now)
```

2.7.24 TimeSerial Function

Returns a **Variant** of subtype **Date** containing the time for a specific hour, minute, and second.

Syntax

TimeSerial(*Hour***, *Minute*, *Second*)**

The **TimeSerial** function syntax has these parts:

Part	Description
Hour	Number between 0 (12:00 A.M.) and 23 (11:00 P. M.), inclusive, or a numeric expression.
Minute	Any numeric expression.
Second	Any numeric expression.

Remarks

To specify a time, such as 11:59:59, the range of numbers for each **TimeSerial** argument should be in the accepted range for the unit; that is, 0–23 for hours and 0–59 for minutes and seconds. However, you can also specify relative times for each argument using any numeric expression that represents some number of hours, minutes, or seconds before or after a certain time.

When any argument exceeds the accepted range for that argument, it increments to the next larger unit as appropriate. For example, if you specify 75 minutes, it is evaluated as one hour and 15 minutes. However, if any single argument is outside the range -32,768 to 32,767, or if the time specified by the three arguments, either directly or by expression, causes the date to fall outside the acceptable range of dates, an error occurs.

Date and Time Functions

Conversion Functions

Example

The following example uses expressions instead of absolute time numbers. The **TimeSerial** function returns a time for 15 minutes before (-15) six hours before noon (12 - 6), or 5:45:00 A.M.

```
Dim MyTime
MyTime = TimeSerial(12 - 6, -15, 0) ' Returns 5:45:00 AM.
```

2.7.25 TimeValue Function

Returns a **Variant** of subtype **Date** containing the time.

Syntax

TimeValue(*Time***)**

The *Time* argument is usually a string expression representing a time from 0:00:00 (12:00:00 A.M.) to 23:59:59 (11:59:59 P.M.), inclusive. However, *Time* can also be any expression that represents a time in that range. If *Time* contains **Null**, **Null** is returned.

Remarks

You can enter valid times using a 12-hour or 24-hour clock. For example, "2:24PM" and "14:24" are both valid *Time* arguments. If the *Time* argument contains date information, **TimeValue** doesn't return the date information. However, if *Time* includes invalid date information, an error occurs.

Date and Time Functions

Conversion Functions

Example

The following example uses the **TimeValue** function to convert a string to a time. You can also use date literals to directly assign a time to a **Variant** (for example, MyTime = #4:35:17 PM#).

```
Dim MyTime
MyTime = TimeValue("4:35:17 PM") ' MyTime contains 4:35:17 PM.
```

2.7.26 UCase Function

Returns a string that has been converted to uppercase.

Syntax

UCase(*String***)**

The *String* argument is any valid string expression. If *String* contains **Null**, **Null** is returned.

Remarks

Only lowercase letters are converted to uppercase; all uppercase letters and nonletter characters remain unchanged.

String Functions

Conversion Functions

Example

The following example uses the **UCase** function to return an uppercase version of a string.

```
Dim MyWord
MyWord = UCase("Hello World") '  Returns "HELLO WORLD".
```

2.7.27 Weekday Function

Returns a whole number representing the day of the week.

Syntax

Weekday(*Date***[, ***FirstDayOfWeek***])**

The **Weekday** function syntax has these parts:

Part	Description
Date	Any expression that can represent a date. If *Date* contains **Null**, **Null** is returned.
FirstDayOfWeek	A constant that specifies the first day of the week. If omitted, **vbSunday** is assumed.

Settings

The *FirstDayOfWeek* argument has these settings:

Constant	Value	Description

vbUseSystem	0	Use National Language Support (NLS) API setting.
vbSunday	1	Sunday
vbMonday	2	Monday
vbTuesday	3	Tuesday
vbWednesday	4	Wednesday
vbThursday	5	Thursday
vbFriday	6	Friday
vbSaturday	7	Saturday

Return Values

The **Weekday** function can return any of these values:

Constant	Value	Description
vbSunday	1	Sunday
vbMonday	2	Monday
vbTuesday	3	Tuesday
vbWednesday	4	Wednesday
vbThursday	5	Thursday
vbFriday	6	Friday
vbSaturday	7	Saturday

Date and Time Functions

Conversion Functions

Example

The following example uses the **Weekday** function to obtain the day of the week from a specified date.

```
Dim MyDate, MyWeekDay
MyDate = #October 19, 1962# ' Assign a date.
' MyWeekDay contains 6 because MyDate represents a Friday.
MyWeekDay = Weekday( MyDate)
```

2.7.28 Year Function

Returns a whole number representing the year.

Syntax

Year(*Date***)**

The *Date* argument is any expression that can represent a date. If *Date* contains **Null**, **Null** is returned.

Date and Time Functions

Conversion Functions

Example

The following example uses the **Year** function to obtain the year from a specified date.

```
Dim MyDate, MyYear
MyDate = #October 19, 1962# ' Assign a date.
MyYear = Year( MyDate)       ' MyYear contains 1962.
```

2.8 Math Functions

- **Abs** Function
- **Atn** Function
- **Cos** Function
- **Exp** Function
- **Fix** Function
- **Int** Function
- **Log** Function
- **Rnd** Function
- **Round** Function
- **Sgn** Function
- **Sin** Function
- **Sqr** Function
- **Tan** Function

2.8.1 Abs Function

Returns the absolute value of a number.

Syntax

Abs(*Number***)**

The *Number* argument can be any valid numeric expression. If *Number* contains **Null**, **Null** is returned; if it is an uninitialized variable, zero is returned.

Remarks

The absolute value of a number is its unsigned magnitude. For example, **Abs(**-1**)** and **Abs (**1**)** both return 1.

Math Functions

Example

The following example uses the **Abs** function to compute the absolute value of a number.

```
Dim MyNumber
MyNumber = Abs( 50.3)   ' Returns 50.3.
MyNumber = Abs( -50.3) ' Returns 50.3.
```

2.8.2 Atn Function

Returns the arctangent of a number.

Syntax

Atn(*Number***)**

The *Number* argument can be any valid numeric expression.

Remarks

The **Atn** function takes the ratio of two sides of a right triangle (*Number*) and returns the corresponding angle in radians. The ratio is the length of the side opposite the angle divided by the length of the side adjacent to the angle. The range of the result is -pi/2 to pi/2 radians.

To convert degrees to radians, multiply degrees by pi/180. To convert radians to degrees, multiply radians by 180/pi.

NOTE

Atn is the inverse trigonometric function of Tan, which takes an angle as its argument and returns the ratio of two sides of a right triangle. Do not confuse **Atn** with the cotangent, which is the simple inverse of a tangent (1/tangent).

Math Functions

Example

The following example uses **Atn** to calculate the value of pi.

```
Dim pi
pi = 4 * Atn(1) ' Calculate the value of pi.
```

2.8.3 Cos Function

Returns the cosine of an angle.

Syntax

Cos(*Number***)**

The *Number* argument can be any valid numeric expression that expresses an angle in radians.

Remarks

The **Cos** function takes an angle and returns the ratio of two sides of a right triangle. The ratio is the length of the side adjacent to the angle divided by the length of the hypotenuse. The result lies in the range -1 to 1.

To convert degrees to radians, multiply degrees by pi/180. To convert radians to degrees, multiply radians by 180/pi.

Math Functions

Example

The following example uses the **Cos** function to return the cosine of an angle.

```
Dim MyAngle, MySecant
MyAngle = 1.3                ' Define angle in radians.
MySecant = 1 / Cos( MyAngle) ' Calculate secant.
```

2.8.4 Exp Function

Returns *e* (the base of natural logarithms) raised to a power.

Syntax

Exp(*Number***)**

The *Number* argument can be any valid numeric expression.

Remarks

If the value of *Number* exceeds 709.782712893, an error occurs. The constant *e* is approximately 2.718282.

NOTE

The **Exp** function complements the action of the Log function and is sometimes referred to as the antilogarithm.

 Math Functions

Example

The following example uses the **Exp** function to return e raised to a power.

```
Dim MyAngle, MyHSin
MyAngle = 1.3 ' Define angle in radians.
' Calculate hyperbolic sine.
MyHSin = (Exp(MyAngle) - Exp(-1 * MyAngle)) / 2
```

2.8.5 Fix Function

Returns the integer portion of a number.

Syntax

Fix(*Number***)**

The *Number* argument can be any valid numeric expression. If *Number* contains **Null**, **Null** is returned.

Remarks

Both Int and **Fix** remove the fractional part of *Number* and return the resulting integer value.

The difference between Int and **Fix** is that if *Number* is negative, Int returns the first negative integer less than or equal to *Number,* whereas **Fix** returns the first negative integer greater than or equal to *Number.* For example, Int converts -8.4 to -9, and **Fix** converts -8.4 to -8.

Fix(*Number*) is equivalent to:

```
Sgn( Number) * Int( Abs( Number))
```

Data Type Functions

Conversion Functions

Math Functions

Example

The following examples illustrate how the Int and **Fix** functions return integer portions of numbers.

```
Dim MyNumber
MyNumber = Int( 99.8)   ' Returns 99.
MyNumber = Fix( 99.2)   ' Returns 99.
MyNumber = Int( -99.8) ' Returns -100.
MyNumber = Fix( -99.8) ' Returns -99.
MyNumber = Int( -99.2) ' Returns -100.
MyNumber = Fix( -99.2) ' Returns -99.
```

2.8.6 Int Function

Returns the integer portion of a number.

Syntax

Int(*Number***)**

The *Number* argument can be any valid numeric expression. If *Number* contains **Null**, **Null** is returned.

Remarks

Both **Int** and Fix remove the fractional part of *Number* and return the resulting integer value.

The difference between **Int** and Fix is that if *Number* is negative, **Int** returns the first negative integer less than or equal to *Number,* whereas Fix returns the first negative integer greater than or equal to *Number.* For example, **Int** converts -8.4 to -9, and Fix converts -8.4 to -8.

Fix(*Number*) is equivalent to.

```
Sgn( Number) * Int( Abs( Number))
```

- Data Type Functions
- Conversion Functions
- Math Functions

Example

The following examples illustrate how the **Int** and Fix functions return integer portions of numbers.

```
Dim MyNumber
MyNumber = Int( 99.8)   ' Returns 99.
MyNumber = Fix( 99.2)   ' Returns 99.
MyNumber = Int(-99.8) ' Returns -100.
MyNumber = Fix(-99.8) ' Returns -99.
MyNumber = Int(-99.2) ' Returns -100.
MyNumber = Fix(-99.2) ' Returns -99.
```

2.8.7 Log Function

Returns the natural logarithm of a number.

Syntax

Log(*Number***)**

The *Number* argument can be any valid numeric expression greater than 0.

Remarks

The natural logarithm is the logarithm to the base *e*. The constant *e* is approximately 2.718282.

You can calculate base-*n* logarithms for any number *x* by dividing the natural logarithm of *x* by the natural logarithm of *n* as follows.

```
Logn( x)  = Log( x)  / Log( n)
```

- Math Functions

Example

The following example illustrates a custom function that calculates base-10 logarithms.

```
Function Log10( X)
  Log10 = Log( X) / Log( 10)
End Function
```

2.8.8 Rnd Function

Returns a random number.

Syntax

Rnd*[(Number)]*

The *Number* argument can be any valid numeric expression.

Remarks

The **Rnd** function returns a value less than 1 but greater than or equal to 0. The value of *Number* determines how **Rnd** generates a random number:

If **Number** *is*	**Rnd generates**
Less than zero	The same number every time, using *Number* as the seed.
Greater than zero	The next random number in the sequence.
Equal to zero	The most recently generated number.
Not supplied	The next random number in the sequence.

For any given initial seed, the same number sequence is generated because each successive call to the **Rnd** function uses the previous number as a seed for the next number in the sequence.

Before calling **Rnd**, use the Randomize statement without an argument to initialize the random-number generator with a seed based on the system timer.

NOTE

To repeat sequences of random numbers, call **Rnd** with a negative argument immediately before using Randomize with a numeric argument. Using Randomize with the same value for *Number* does not repeat the previous sequence.

Math Functions

Example

To produce random integers in a given range, use this formula (here, upperbound is the highest number in the range, and lowerbound is the lowest number in the range).

```
Int(( upperbound - lowerbound + 1) * Rnd + lowerbound)
```

2.8.9 Round Function

Returns a number rounded to a specified number of decimal places.

Syntax

Round(*Expression[***, *NumDecimalPlaces]***)**

The **Round** function syntax has these parts:

Part	Description
Expression	Required. Numeric expression being rounded.
NumDecimalPlaces	Optional. Number indicating how many places to the right of the decimal are included in the rounding. If omitted, integers are returned by the **Round** function.

Math Functions

Example

The following example uses the **Round** function to round a number to two decimal places.

```
Dim MyVar, pi
pi = 3.14159
MyVar = Round( pi, 2) ' MyVar contains 3.14.
```

2.8.10 Sgn Function

Returns an integer indicating the sign of a number.

Syntax

Sgn(*Number***)**

The *Number* argument can be any valid numeric expression.

Return Values

The **Sgn** function has the following return values:

If **Number** *is*	**Sgn returns**
Greater than zero	1
Equal to zero	0
Less than zero	-1

Remarks

The sign of the *Number* argument determines the return value of the **Sgn** function.

Math Functions

Example

The following example uses the **Sgn** function to determine the sign of a number.

```
Dim MyVar1, MyVar2, MyVar3, MySign
MyVar1 = 12
MyVar2 = -2.4
MyVar3 = 0
MySign = Sgn(MyVar1) ' Returns 1.
MySign = Sgn(MyVar2) ' Returns -1.
MySign = Sgn(MyVar3) ' Returns 0.
```

2.8.11 Sin Function

Returns the sine of an angle.

Syntax

Sin(*Number***)**

The *Number* argument can be any valid numeric expression that expresses an angle in radians.

Remarks

The **Sin** function takes an angle and returns the ratio of two sides of a right triangle. The ratio is the length of the side opposite the angle divided by the length of the hypotenuse. The result lies in the range -1 to 1.

To convert degrees to radians, multiply degrees by pi/180. To convert radians to degrees, multiply radians by 180/pi.

Math Functions

Example

The following example uses the **Sin** function to return the sine of an angle.

```
Dim MyAngle, MyCosecant
MyAngle = 1.3                  ' Define angle in radians.
MyCosecant = 1 / Sin(MyAngle) ' Calculate cosecant.
```

2.8.12 Sqr Function

Returns the square root of a number.

Syntax

Sqr(Number**)**

The *Number* argument can be any valid numeric expression greater than or equal to 0.

Math Functions

Example

The following example uses the **Sqr** function to calculate the square root of a number.

```
Dim MySqr
MySqr = Sqr(4)   ' Returns 2.
MySqr = Sqr(23)  ' Returns 4.79583152331272.
MySqr = Sqr(0)   ' Returns 0.
MySqr = Sqr(-4)  ' Generates a run-time error.
```

2.8.13 Tan Function

Returns the tangent of an angle.

Syntax

Tan(*Number***)**

The *Number* argument can be any valid numeric expression that expresses an angle in radians.

Remarks

Tan takes an angle and returns the ratio of two sides of a right triangle. The ratio is the length of the side opposite the angle divided by the length of the side adjacent to the angle.

To convert degrees to radians, multiply degrees by pi/180. To convert radians to degrees, multiply radians by 180/pi.


Math Functions

Example

The following example uses the **Tan** function to return the tangent of an angle.

```
Dim MyAngle, MyCotangent
MyAngle = 1.3                    ' Define angle in radians.
MyCotangent = 1 / Tan(MyAngle)   ' Calculate cotangent.
```

2.9 Miscellaneous Functions

- **Eval** Function
- **GetObject** Function
- **GetRef** Function
- **InputBox** Function
- **LoadPicture** Function
- **MsgBox** Function
- **RGB** Function
- **ScriptEngine** Function
- **ScriptEngineBuildVersion** Function
- **ScriptEngineMajorVersion** Function
- **ScriptEngineMinorVersion** Function

2.9.1 Eval Function

Evaluates an expression and returns the result.

Syntax

[Result =] **Eval(***Expression***)**

The **Eval** function syntax has these parts:

Part	Description
Result	Optional. Variable to which return value assignment is made. If *Result* is not specified, consider using the Execute statement instead.
Expression	Required. String containing any legal *VBScript* expression.

Remarks

In *VBScript*, *x* = *y* can be interpreted two ways. The first is as an assignment statement, where the value of *y* is assigned to *x*. The second interpretation is as an expression that tests if *x* and *y* have the same value. If they do, *Result* is **True**; if they are not, *Result* is **False**. The **Eval** method always uses the second interpretation, whereas the Execute statement always uses the first.

Miscellaneous Functions

Example

The following example illustrates the use of the **Eval** function.

```
Sub GuessANumber
  Dim Guess, RndNum
  RndNum = Int((100) * Rnd(1) + 1)
  Guess = CInt(InputBox("Enter your guess:", , 0))
  Do
    If Eval("Guess = RndNum") Then
      MsgBox "Congratulations! You guessed it!"
      Exit Sub
    Else
      Guess = CInt(InputBox("Sorry! Try again.", , 0))
    End If
  Loop Until Guess = 0
End Sub
```

2.9.2 GetObject Function

Returns a reference to an Automation object from a file.

Syntax

GetObject([*PathName***][, *Class*])**

The **GetObject** function syntax has these parts:

Part	Description

PathName	Optional; **String**. Full path and name of the file containing the object to retrieve. If *PathName* is omitted, *Class* is required.
Class	Optional; **String**. Class of the object.

The *Class* argument uses the syntax *AppName.ObjectType* and has these parts:

Part	Description
AppName	Required; **String**. Name of the application providing the object.
ObjectType	Required; **String**. Type or class of object to create.

Remarks

Use the **GetObject** function to access an Automation object from a file and assign the object to an object variable. Use the Set statement to assign the object returned by **GetObject** to the object variable. For Example:

```
Dim CADObject
Set CADObject = GetObject("C:\CAD\SCHEMA.CAD")
```

When this code is executed, the application associated with the specified pathname is started and the object in the specified file is activated. If *PathName* is a zero-length string (""), **GetObject** returns a new object instance of the specified type. If the *PathName* argument is omitted, **GetObject** returns a currently active object of the specified type. If no object of the specified type exists, an error occurs.

Some applications allow you to activate part of a file. Add an exclamation point (!) to the end of the file name and follow it with a string that identifies the part of the file you want to activate. For information on how to create this string, see the documentation for the application that created the object.

For example, in a drawing application you might have multiple layers to a drawing stored in a file. You could use the following code to activate a layer within a drawing called p`ebj ^K^a:

```
Set LayerObject = GetObject("C:\CAD\SCHEMA.CAD!Layer3")
```

If you don't specify the object's class, Automation determines the application to start and the object to activate, based on the file name you provide. Some files, however, may support more than one class of object. For example, a drawing might support three different types of objects: an Application object, a Drawing object, and a Toolbar object, all of which are part of the same file. To specify which object in a file you want to activate, use the optional *Class* argument. For Example:

```
Dim MyObject
Set MyObject = GetObject("C:\DRAWINGS\SAMPLE.DRW", "FIGMENT.DRAWING")
```

In the preceding example, cfdj bkq is the name of a drawing application and ao^t fkd is one of the object types it supports. Once an object is activated, you reference it in code using the object variable you defined. In the preceding example, you access properties and methods of the new object using the object variable j ól ÄáÉÅí. For Example:

```
MyObject.Line 9, 90
MyObject.InsertText 9, 100, "Hello, world."
MyObject.SaveAs "C:\DRAWINGS\SAMPLE.DRW"
```

NOTE

Use the **GetObject** function when there is a current instance of the object or if you want to create the object with a file already loaded. If there is no current instance, and you don't want the object started with a file loaded, use the CreateObject function.

If an object has registered itself as a single-instance object, only one instance of the object is created, no matter how many times CreateObject is executed. With a single-instance object, **GetObject** always returns the same instance when called with the zero-length string ("") syntax, and it causes an error if the *PathName* argument is omitted.

Miscellaneous Functions

2.9.3 GetRef Function

Returns a reference to a procedure that can be bound to an event.

Syntax

Set *Object.EventName* **= GetRef(***ProcName***)**

The **GetRef** function syntax has these parts:

Part	Description
Object	Required. Name of the object with which *EventName* is associated.
EventName	Required. Name of the event to which the function is to be bound.
ProcName	Required. String containing the name of the Sub or Function procedure being associated with the *EventName*.

Remarks

The **GetRef** function allows you to connect a *VBScript* procedure (Function or Sub) to any available event on your DHTML (Dynamic HTML) pages. The DHTML object model provides information about what events are available for its various objects.

In other scripting and programming languages, the functionality provided by **GetRef** is referred to as a function pointer, that is, it points to the address of a procedure to be executed when the specified event occurs.

Miscellaneous Functions

Example

The following example illustrates the use of the **GetRef** function.

```
<SCRIPT LANGUAGE="VBScript">
Function GetRefTest()
  Dim Splash
  Splash = "GetRefTest Version 1.0"  & vbCrLf
  Splash = Splash & Chr(169) & " GEOVAP - Reliance "
  MsgBox Splash
End Function
Set Window.Onload = GetRef("GetRefTest")
</SCRIPT>
```

2.9.4 InputBox Function

Displays a prompt in a dialog box, waits for the user to input text or click a button, and returns the contents of the text box.

Syntax

InputBox(*Prompt***[, ***Title***][, ***Default***][, ***XPos***][, ***YPos***][, ***HelpFile***, ***Context***])**

The **InputBox** function syntax has these parts:

Part	Description
Prompt	String expression displayed as the message in the dialog box. The maximum length of *Prompt* is approximately 1024 characters, depending on the width of the characters used. If *Prompt* consists of more than one line, you can separate the lines using a carriage return character (**Chr(**13**)**), a linefeed character (**Chr(**10**)**), or carriage return–linefeed character combination (**Chr(**13**) & Chr(**10**)**) between each line.
Title	String expression displayed in the title bar of the dialog box. If you omit *Title*, the application name is placed in the title bar.
Default	String expression displayed in the text box as the default response if no other input is provided. If you omit *Default*, the text box is displayed empty.
XPos	Numeric expression that specifies, in twips, the horizontal distance of the left edge of the dialog box from the left edge of the screen. If *XPos* is omitted, the dialog box is horizontally centered.
YPos	Numeric expression that specifies, in twips, the vertical distance of the upper edge of the dialog box from the top of the screen. If *YPos* is omitted, the dialog box is vertically positioned approximately one-third of the way down the screen.
HelpFile	String expression that identifies the Help file to use to provide context-sensitive Help for the dialog box. If *HelpFile* is provided, *Context* must also be provided.

Context	Numeric expression that identifies the Help context number assigned by the Help author to the appropriate Help topic. If *Context* is provided, *HelpFile* must also be provided.

Remarks

When both *HelpFile* and *Context* are supplied, a Help button is automatically added to the dialog box.

If the user clicks *OK* or presses *Enter*, the **InputBox** function returns whatever is in the text box. If the user clicks *Cancel*, the function returns a zero-length string ("").

Miscellaneous Functions

Example

The following example uses the **InputBox** function to display an input box and assign the string to the variable Input.

```
Dim Input
Input = InputBox("Enter your name")
MsgBox ("You entered: " & Input)
```

2.9.5 LoadPicture Function

Returns a picture object. Available only on 32-bit platforms.

Syntax

LoadPicture(*PictureName***)**

The *PictureName* argument is a string expression that indicates the name of the picture file to be loaded.

Remarks

Graphics formats recognized by **LoadPicture** include bitmap (*.bmp*) files, icon (*.ico*) files, run-length encoded (*.rle*) files, metafile (*.wmf*) files, enhanced metafiles (*.emf*), GIF (*.gif*) files, and JPEG (*.jpg*) files.

Miscellaneous Functions

2.9.6 MsgBox Function

Displays a message in a dialog box, waits for the user to click a button, and returns a value indicating which button the user clicked.

Syntax

MsgBox(*Prompt[***, ***Buttons][***, ***Title][***, ***HelpFile***, ***Context]***)**

The **MsgBox** function syntax has these parts:

Part	Description
Prompt	String expression displayed as the message in the dialog box. The maximum length of *Prompt* is approximately 1024 characters, depending on the width of the characters used. If *Prompt* consists of more than one line, you can separate the lines using a carriage return character (**Chr(**13**)**), a linefeed character (**Chr(**10**)**), or carriage return–linefeed character combination (**Chr(**13**) & Chr(**10**)**) between each line.
Buttons	Numeric expression that is the sum of values specifying the number and type of buttons to display, the icon style to use, the identity of the default button, and the modality of the message box. See Settings section for values. If omitted, the default value for *Buttons* is 0.
Title	String expression displayed in the title bar of the dialog box. If you omit *Title*, the application name is placed in the title bar.
HelpFile	String expression that identifies the Help file to use to provide context-sensitive Help for the dialog box. If *HelpFile* is provided, *Context* must also be provided. Not available on 16-bit platforms.
Context	Numeric expression that identifies the Help context number assigned by the Help author to the appropriate Help topic. If *Context* is provided, *HelpFile* must also be provided. Not available on 16-bit platforms.

Settings

The *Buttons* argument settings are:

Constant	Value	Description
vbOKOnly	0	Display *OK* button only.
vbOKCancel	1	Display *OK* and *Cancel* buttons.
vbAbortRetryIgnore	2	Display *Abort*, *Retry*, and *Ignore* buttons.
vbYesNoCancel	3	Display *Yes*, *No*, and *Cancel* buttons.
vbYesNo	4	Display *Yes* and *No* buttons.
vbRetryCancel	5	Display *Retry* and *Cancel* buttons.
vbCritical	16	Display *Critical Message* icon.
vbQuestion	32	Display *Warning Query* icon.
vbExclamation	48	Display *Warning Message* icon.
vbInformation	64	Display *Information Message* icon.
vbDefaultButton1	0	First button is default.
vbDefaultButton2	256	Second button is default.
vbDefaultButton3	512	Third button is default.
vbDefaultButton4	768	Fourth button is default.
vbApplicationModal	0	Application modal; the user must respond to the message box before continuing work in the current application.
vbSystemModal	4096	System modal; all applications are suspended until the user responds to the message box.

The first group of values (0–5) describes the number and type of buttons displayed in the dialog box; the second group (16, 32, 48, 64) describes the icon style; the third group (0, 256, 512, 768) determines which button is the default; and the fourth group (0, 4096) determines the modality of the message box. When adding numbers to create a final value for the argument *Buttons*, use only one number from each group.

Return Values

The **MsgBox** function has the following return values:

Constant	Value	Button
vbOK	1	*OK*
vbCancel	2	*Cancel*
vbAbort	3	*Abort*
vbRetry	4	*Retry*
vbIgnore	5	*Ignore*
vbYes	6	*Yes*
vbNo	7	*No*

Remarks

When both *HelpFile* and *Context* are provided, the user can press *F1* to view the Help topic corresponding to the context.

If the dialog box displays a *Cancel* button, pressing the *Esc* key has the same effect as clicking *Cancel*. If the dialog box contains a *Help* button, context-sensitive Help is provided for the dialog box. However, no value is returned until one of the other buttons is clicked.

Miscellaneous Functions

Example

The following example uses the **MsgBox** function to display a message box and return a value describing which button was clicked.

```
Dim MyVar
```

```
' MyVar contains either 1 or 2, depending on which button is clicked.
MyVar = MsgBox("Hello World!", 65, "MsgBox Example")
```

2.9.7 RGB Function

Returns a whole number representing an RGB color value.

Syntax

RGB(*Red*, *Green*, *Blue***)**

The **RGB** function has these parts:

Part	Description
Red	Required. Number in the range 0–255 representing the red component of the color.
Green	Required. Number in the range 0–255 representing the green component of the color.
Blue	Required. Number in the range 0–255 representing the blue component of the color.

Remarks

Application methods and properties that accept a color specification expect that specification to be a number representing an RGB color value. An RGB color value specifies the relative intensity of red, green, and blue to cause a specific color to be displayed.

The value for any argument to RGB that exceeds 255 is assumed to be 255.

The low-order byte contains the value for red, the middle byte contains the value for green, and the high-order byte contains the value for blue.

For applications that require the byte order to be reversed, the following function will provide the same information with the bytes reversed.

```
Function RevRGB(red, green, blue)
  RevRGB= CLng(blue + (green * 256) + (red * 65536))
End Function
```

Miscellaneous Functions

2.9.8 ScriptEngine Function

Returns a string representing the scripting language in use.

Syntax

```
ScriptEngine
```

Return Values

The **ScriptEngine** function can return any of the following strings:

String	Description
VBScript	Indicates that Microsoft® Visual Basic® Scripting Edition is the current scripting engine.
JScript	Indicates that Microsoft JScript® is the current scripting engine.
VBA	Indicates that Microsoft Visual Basic for Applications is the current scripting engine.

Miscellaneous Functions

Example

The following example uses the **ScriptEngine** function to return a string describing the scripting language in use.

```
Function GetScriptEngineInfo
  Dim s
  s = ""                    ' Build string with necessary info.
  s = ScriptEngine & " Version "
  s = s & ScriptEngineMajorVersion & "."
  s = s & ScriptEngineMinorVersion & "."
  s = s & ScriptEngineBuildVersion
  GetScriptEngineInfo = s ' Return the results.
End Function
```

2.9.9 ScriptEngineBuildVersion Function

Returns the build version number of the scripting engine in use.

Syntax

```
ScriptEngineBuildVersion
```

Remarks

The return value corresponds directly to the version information contained in the DLL for the scripting language in use.

Miscellaneous Functions

Example

The following example uses the **ScriptEngineBuildVersion** function to return the build version number of the scripting engine.

```
Function GetScriptEngineInfo
  Dim s
  s = ""                  ' Build string with necessary info.
  s = ScriptEngine & " Version "
  s = s & ScriptEngineMajorVersion & "."
  s = s & ScriptEngineMinorVersion & "."
  s = s & ScriptEngineBuildVersion
  GetScriptEngineInfo = s ' Return the results.
End Function
```

2.9.10 ScriptEngineMajorVersion Function

Returns the major version number of the scripting engine in use.

Syntax

```
ScriptEngineMajorVersion
```

Remarks

The return value corresponds directly to the version information contained in the DLL for the scripting language in use.

Miscellaneous Functions

Example

The following example uses the **ScriptEngineMajorVersion** function to return the version number of the scripting engine.

```
Function GetScriptEngineInfo
  Dim s
  s = ""                  ' Build string with necessary info.
  s = ScriptEngine & " Version "
  s = s & ScriptEngineMajorVersion & "."
  s = s & ScriptEngineMinorVersion & "."
  s = s & ScriptEngineBuildVersion
  GetScriptEngineInfo = s ' Return the results.
End Function
```

2.9.11 ScriptEngineMinorVersion Function

Returns the minor version number of the scripting engine in use.

Syntax

```
ScriptEngineMinorVersion
```

Remarks

The return value corresponds directly to the version information contained in the DLL for the scripting language in use.

Miscellaneous Functions

Example

The following example uses the **ScriptEngineMinorVersion** function to return the minor version number of the scripting engine.

```
Function GetScriptEngineInfo
  Dim s
  s = ""                  ' Build string with necessary info.
  s = ScriptEngine & " Version "
```

```
  s = s & ScriptEngineMajorVersion & "."
  s = s & ScriptEngineMinorVersion & "."
  s = s & ScriptEngineBuildVersion
  GetScriptEngineInfo = s ' Return the results.
End Function
```

2.10 VBScript Statements

- **Call** Statement
- **Const** Statement
- **Dim** Statement
- **Do...Loop** Statement
- **Erase** Statement
- **Execute** Statement
- **Exit** Statement
- **For Each...Next** Statement
- **For...Next** Statement
- **Function** Statement
- **If...Then...Else** Statement
- **On Error** Statement
- **Option Explicit** Statement
- **Private** Statement
- **Public** Statement
- **Randomize** Statement
- **ReDim** Statement
- **Rem** Statement
- **Select Case** Statement
- **Set** Statement
- **Stop** Statement
- **Sub** Statement
- **While...WEnd** Statement
- **With** Statement

2.10.1 Call Statement

Transfers control to a Sub or Function procedure.

Syntax

*[**Call**] Name [ArgumentList]*

The **Call** statement syntax has these parts:

Part	Description
Call	Optional keyword. If specified, you must enclose *ArgumentList* in parentheses. For Example: `Call MyProc(0)`
Name	Required. Name of the procedure to call.
ArgumentList	Optional. Comma-delimited list of variables, arrays, or expressions to pass to the procedure.

Remarks

You are not required to use the **Call** keyword when calling a procedure. However, if you use the **Call** keyword to call a procedure that requires arguments, *ArgumentList* must be enclosed in parentheses. If you omit the **Call** keyword, you also must omit the parentheses around *ArgumentList*. If you use either **Call** syntax to call any intrinsic or user-defined function, the function's return value is discarded.

VBScript Statements

Example

```
Call MyFunction("Hello World")
Function MyFunction(text)
  MsgBox text
End Function
```

2.10.2 Const Statement

Declares constants for use in place of literal values.

Syntax

[**Public** | **Private**] **Const** *ConstName* = *Expression*

The **Const** statement syntax has these parts:

Part	Description
Public	Optional. Keyword used at a script level to declare constants that are available to all procedures in all scripts. Not allowed in procedures.
Private	Optional. Keyword used at a script level to declare constants that are available only within the script where the declaration is made. Not allowed in procedures.
ConstName	Required. Name of the constant; follows standard variable naming conventions.
Expression	Required. Literal or other constant, or any combination that includes all arithmetic or logical operators except Is.

Remarks

Constants are public by default. Within procedures, constants are always private; their visibility can't be changed. Within a script, the default visibility of a script-level constant can be changed using the Private keyword.

To combine several constant declarations on the same line, separate each constant assignment with a comma. When constant declarations are combined in this way, the Public or Private keyword, if used, applies to all of them.

You can't use variables, user-defined functions, or intrinsic *VBScript* functions (such as Chr) in constant declarations. By definition, they can't be constants. You also can't create a constant from any expression that involves an operator, that is, only simple constants are allowed. Constants declared in a Sub or Function procedure are local to that procedure. A constant declared outside a procedure is defined throughout the script in which it is declared. You can use constants anywhere you can use an expression.

NOTE

Constants can make your scripts self-documenting and easy to modify. Unlike variables, constants can't be inadvertently changed while your script is running.

VBScript Statements

Example

The following code illustrates the use of the **Const** statement.

```
Const MyVar = 459 ' Constants are Public by default.
' Declare Private constant.
Private Const MyString = "HELP"
' Declare multiple constants on same line.
Const MyStr = "Hello", MyNumber = 3.4567
```

2.10.3 Dim Statement

Declares variables and allocates storage space.

Syntax

Dim *VarName*[**(**[*Subscripts*]**)**][**,** *VarName*[**(**[*Subscripts*]**)**]]...

The **Dim** statement syntax has these parts:

Part	Description
VarName	Name of the variable; follows standard variable naming conventions.

Subscripts	Dimensions of an array variable; up to 60 multiple dimensions may be declared. The *Subscripts* argument uses the following syntax: *upperbound[, upperbound]...* The lower bound of an array is always zero.

Remarks

Variables declared with **Dim** at the script level (globally) are available to all procedures in all scripts within the same thread (similar to **Public** statement). At the procedure level, variables are available only within the procedure.

You can also use the **Dim** statement with empty parentheses to declare a dynamic array. After declaring a dynamic array, use the ReDim statement within a procedure to define the number of dimensions and elements in the array. If you try to redeclare a dimension for an array variable whose size was explicitly specified in a **Dim** statement, an error occurs.

TIP

When you use the **Dim** statement in a procedure, you generally put the **Dim** statement at the beginning of the procedure.

Array Functions

VBScript Statements

Example

The following examples illustrate the use of the **Dim** statement.

```
Dim Names(9)      ' Declare an array with 10 elements.
Dim Names()       ' Declare a dynamic array.
Dim MyVar, MyNum ' Declare two variables.
```

2.10.4 Do...Loop Statement

Repeats a block of statements while a condition is **True** or until a condition becomes **True**.

Syntax

```
Do [{While | Until} Condition]
  [Statements]
  [Exit Do]
  [Statements]
Loop
```

Or, you can use this syntax:

```
Do
  [Statements]
  [Exit Do]
  [Statements]
Loop [{While | Until} Condition]
```

The **Do...Loop** statement syntax has these parts:

Part	Description
Condition	Numeric or string expression that is **True** or **False**. If *Condition* is **Null**, *Condition* is treated as **False**.
Statements	One or more statements that are repeated while or until *Condition* is **True**.

Remarks

The Exit **Do** can only be used within a **Do...Loop** control structure to provide an alternate way to exit a **Do...Loop**. Any number of Exit **Do** statements may be placed anywhere in the **Do...Loop**. Often used with the evaluation of some condition (for example, If...Then), Exit **Do** transfers control to the statement immediately following the **Loop**.

When used within nested **Do...Loop** statements, Exit **Do** transfers control to the loop that is nested one level above the loop where it occurs.

VBScript Statements

Example

The following examples illustrate use of the **Do...Loop** statement.

```
Do Until DefResp = vbNo
  ' Generate a random integer between 1 and 6.
  MyNum = Int ( 6 * Rnd + 1)
  DefResp = MsgBox ( MyNum & " Do you want another number?", vbYesNo)
Loop
Dim Check, Counter
Check = True
Counter = 0                ' Initialize variables.
Do                         ' Outer loop.
  Do While Counter < 20    ' Inner loop.
    Counter = Counter + 1 ' Increment Counter.
    If Counter = 10 Then   ' If condition is True...
      Check = False        ' set value of flag to False.
      Exit Do              ' Exit inner loop.
    End If
  Loop
Loop Until Check = False   ' Exit outer loop immediately.
```

2.10.5 Erase Statement

Reinitializes the elements of fixed-size arrays and deallocates dynamic-array storage space.

Syntax

Erase *Array*

The *Array* argument is the name of the array variable to be erased.

Remarks

It is important to know whether an array is fixed-size (ordinary) or dynamic because **Erase** behaves differently depending on the type of array. **Erase** recovers no memory for fixed-size arrays. **Erase** sets the elements of a fixed array as follows:

Type of array	Effect of Erase on fixed-array elements
Fixed numeric array	Sets each element to zero.
Fixed string array	Sets each element to zero-length ("").

Array of objects	Sets each element to the special value **Nothing**.

Erase frees the memory used by dynamic arrays. Before your program can refer to the dynamic array again, it must redeclare the array variable's dimensions using a ReDim statement.

Array Functions

VBScript Statements

Example

The following example illustrates the use of the **Erase** statement.

```
Dim NumArray(9)
Dim DynamicArray()
ReDim DynamicArray(9) ' Allocate storage space.
Erase NumArray        ' Each element is reinitialized.
Erase DynamicArray    ' Free memory used by array.
```

2.10.6 Execute Statement

Executes one or more specified statements.

Syntax

Execute Statement

The required *Statement* argument is a string expression containing one or more statements for execution. Include multiple statements in the *Statement* argument, using colons or embedded line breaks to separate them.

Remarks

In *VBScript*, *x* = *y* can be interpreted two ways. The first is as an assignment statement, where the value of *y* is assigned to *x*. The second interpretation is as an expression that tests if *x* and *y* have the same value. If they do, *Result* is **True**; if they are not, *Result* is **False**. The **Execute** statement always uses the first interpretation, whereas the Eval method always uses the second.

The context in which the **Execute** statement is invoked determines what objects and variables are available to the code being run. In-scope objects and variables are available to code running in an **Execute** statement. However, it is important to understand that if you execute code that creates a procedure, that procedure does not inherit the scope of the procedure in which it occurred.

Like any procedure, the new procedure's scope is global, and it inherits everything in the global scope. Unlike any other procedure, its context is not global scope, so it can only be executed in the context of the procedure where the **Execute** statement occurred. However, if the same **Execute** statement is invoked outside of a procedure (i.e., in global scope), not only does it inherit everything in global scope, but it can also be called from anywhere, since its context is global.

 *VBScript* Statements

Example

The following example shows how the **Execute** statement can be rewritten so you don't have to enclose the entire procedure in the quotation marks.

```
S = "Sub Proc2" & vbCrLf
S = S & "  Print X" & vbCrLf
S = S & "End Sub"
Execute S
```

2.10.7 Exit Statement

Exits a block of Do...Loop, For Each...Next, For...Next, Function, or Sub code.

Syntax

```
Exit Do

Exit For

Exit Function

Exit Sub
```

The **Exit** statement syntax has these forms:

Statement	Description

Exit Do	Provides a way to exit a Do...Loop statement. It can be used only inside a Do...Loop statement. **Exit Do** transfers control to the statement following the **Loop** statement. When used within nested Do...Loop statements, Exit **Do** transfers control to the loop that is one nested level above the loop where it occurs.
Exit For	Provides a way to exit a **For** loop. It can be used only in a For...Next or For Each...Next loop. **Exit For** transfers control to the statement following the **Next** statement. When used within nested **For** loops, **Exit For** transfers control to the loop that is one nested level above the loop where it occurs.
Exit Function	Immediately exits the Function procedure in which it appears. Execution continues with the statement following the statement that called the Function.
Exit Sub	Immediately exits the Sub procedure in which it appears. Execution continues with the statement following the statement that called the Sub.

VBScript Statements

Example

The following example illustrates the use of the **Exit** statement.

```
Sub RandomLoop
  Dim I, MyNum
  Do                          ' Set up infinite loop.
    For I = 1 To 1000         ' Loop 1000 times.
      MyNum = Int( Rnd * 100) ' Generate random numbers.
      Select Case MyNum       ' Evaluate random number.
        Case 17: MsgBox "Case 17"
          Exit For            ' If 17, exit For...Next.
        Case 29: MsgBox "Case 29"
          Exit Do             ' If 29, exit Do...Loop.
        Case 54: MsgBox "Case 54"
          Exit Sub            ' If 54, exit Sub procedure.
      End Select
    Next
```

```
    Loop
End Sub
```

2.10.8 For Each...Next Statement

Repeats a group of statements for each element in an array or collection.

Syntax

For Each *Element* **In** *Group*

[Statements]

*[***Exit For***]*

[Statements]

Next *[Element]*

The **For Each...Next** statement syntax has these parts:

Part	Description
Element	Variable used to iterate through the elements of the collection or array. For collections, *Element* can only be a **Variant** variable, a generic **Object** variable, or any specific Automation object variable. For arrays, *Element* can only be a **Variant** variable.
Group	Name of an object collection or array.
Statements	One or more statements that are executed on each item in *Group*.

Remarks

The **For Each** block is entered if there is at least one element in *Group*. Once the loop has been entered, all the statements in the loop are executed for the first element in *Group*. As long as there are more elements in *Group*, the statements in the loop continue to execute for each element. When there are no more elements in *Group*, the loop is exited and execution continues with the statement following the **Next** statement.

The Exit **For** can only be used within a **For Each...Next** or For...Next control structure to provide an alternate way to exit. Any number of Exit **For** statements may be placed anywhere in the loop. The Exit **For** is often used with the evaluation of some condition (for example, If...Then), and transfers control to the statement immediately following **Next**.

You can nest **For Each...Next** loops by placing one **For Each...Next** loop within another. However, each loop *Element* must be unique.

NOTE

If you omit *Element* in a **Next** statement, execution continues as if you had included it. If a **Next** statement is encountered before its corresponding **For** statement, an error occurs.

VBScript Statements

Example

The following example illustrates use of the **For Each...Next** statement.

```
Function ShowFolderList(folderspec)
  Dim fso, f, f1, fc, s
  Set fso = CreateObject("Scripting.FileSystemObject")
  Set f = fso.GetFolder(folderspec)
  Set fc = f.Files
  For Each f1 In fc
    s = s & f1.name
    s = s & "<BR>"
  Next  ShowFolderList = s
End Function
```

2.10.9 For...Next Statement

Repeats a group of statements a specified number of times.

Syntax

```
For Counter = Start To End [Step Step]
  [Statements]
  [Exit For]
```

```
    [ Statements]

Next
```

The **For...Next** statement syntax has these parts:

Part	Description
Counter	Numeric variable used as a loop counter. The variable can't be an array element or an element of a user-defined type.
Start	Initial value of *Counter*.
End	Final value of *Counter*.
Step	Amount *Counter* is changed each time through the loop. If not specified, *Step* defaults to one.
Statements	One or more statements between **For** and **Next** that are executed the specified number of times.

Remarks

The *Step* argument can be either positive or negative. The value of the *Step* argument determines loop processing as follows:

Value	Loop executes if
Positive or 0	*Counter* <= *End*
Negative	*Counter* >= *End*

Once the loop starts and all statements in the loop have executed, *Step* is added to *Counter*. At this point, either the statements in the loop execute again (based on the same test that caused the loop to execute initially), or the loop is exited and execution continues with the statement following the **Next** statement.

TIP

Changing the value of *Counter* while inside a loop can make it more difficult to read and debug your code.

The Exit **For** can only be used within a For Each...Next or **For...Next** control structure to provide an alternate way to exit. Any number of Exit **For** statements may be placed anywhere in the loop. The Exit **For** is often used with the evaluation of some condition (for example, If...Then), and transfers control to the statement immediately following **Next**.

You can nest **For...Next** loops by placing one **For...Next** loop within another. Give each loop a unique variable name as its *Counter*. The following construction is correct.

```
For I = 1 To 10
  For J = 1 To 10
    For K = 1 To 10
      ' ...
    Next
  Next
Next
```

VBScript Statements

2.10.10 Function Statement

Declares the name, arguments, and code that form the body of a **Function** procedure.

Syntax

[**Public** | **Private**] **Function** *Name [*(*ArgList*)*]*

[Statements]

[Name = Expression]

[Exit Function]

[Statements]

[Name = Expression]

End Function

The **Function** statement syntax has these parts:

Part	Description

Public	Indicates that the **Function** procedure is accessible to all other procedures in all scripts.
Private	Indicates that the **Function** procedure is accessible only to other procedures in the script where it is declared or if the function is a member of a class, and that the **Function** procedure is accessible only to other procedures in that class.
Name	Name of the **Function**; follows standard variable naming conventions.
ArgList	List of variables representing arguments that are passed to the **Function** procedure when it is called. Multiple variables are separated by commas.
Statements	Any group of statements to be executed within the body of the **Function** procedure.
Expression	Return value of the **Function**.

The *ArgList* argument has the following syntax and parts:

[**ByVal** | **ByRef**] *VarName*[**( )**]

Part	Description
ByVal	Indicates that the argument is passed by value.
ByRef	Indicates that the argument is passed by reference.
VarName	Name of the variable representing the argument; follows standard variable naming conventions.

Remarks

If not explicitly specified using either Public or Private, **Function** procedures are public by default, that is, they are visible to all other procedures in your script. The value of local variables in a **Function** is not preserved between calls to the procedure.

You can't define a **Function** procedure inside any other procedure (e.g. Sub or Property Get).

The Exit **Function** statement causes an immediate exit from a **Function** procedure. Program execution continues with the statement that follows the statement that called the **Function** procedure. Any number of Exit **Function** statements can appear anywhere in a **Function** procedure.

Like a Sub procedure, a **Function** procedure is a separate procedure that can take arguments, perform a series of statements, and change the values of its arguments. However, unlike a Sub procedure, you can use a Function procedure on the right side of an expression in the same way you use any intrinsic function, such as Sqr, Cos, or Chr, when you want to use the value returned by the function.

You call a **Function** procedure using the function name, followed by the argument list in parentheses, in an expression. See the Call statement for specific information on how to call **Function** procedures.

CAUTION

Function procedures can be recursive, that is, they can call themselves to perform a given task. However, recursion can lead to stack overflow.

To return a value from a function, assign the value to the function name. Any number of such assignments can appear anywhere within the procedure. If no value is assigned to *Name*, the procedure returns a default value: a numeric function returns 0 and a string function returns a zero-length string (""). A function that returns an object reference returns **Nothing** if no object reference is assigned to *Name* (using Set) within the **Function**.

Variables used in **Function** procedures fall into two categories: those that are explicitly declared within the procedure and those that are not. Variables that are explicitly declared in a procedure (using Dim or the equivalent) are always local to the procedure. Variables that are used but not explicitly declared in a procedure are also local unless they are explicitly declared at some higher level outside the procedure.

CAUTION

A procedure can use a variable that is not explicitly declared in the procedure, but a naming conflict can occur if anything you have defined at the script level has the same name. If your procedure refers to an undeclared variable that has the same name as another procedure, constant, or variable, it is assumed that your procedure is referring to that script-level name. To avoid this kind of conflict, use an Option Explicit statement to force explicit declaration of variables.

CAUTION

VBScript may rearrange arithmetic expressions to increase internal efficiency. Avoid using a **Function** procedure in an arithmetic expression when the function changes the value of variables in the same expression.

VBScript Statements

2.10.11 If...Then...Else Statement

Conditionally executes a group of statements, depending on the value of an expression.

Syntax

If *Condition* **Then** *Statements* [**Else** *ElseStatements*]

Or, you can use the block form syntax:

If *Condition* **Then**

[Statements]

[**ElseIf** *Condition-n* **Then**

[ElseIfStatements]]...

[**Else**

[ElseStatements]]

End If

The **If...Then...Else** statement syntax has these parts:

Part	Description
Condition	One or more of the following two types of expressions: A numeric or string expression that evaluates to **True** or **False**. If *Condition* is **Null**, *Condition* is treated as **False**.

	An expression of the form **TypeOf** *ObjectName* **Is** *Objecttype*. The *ObjectName* is any object reference and *Objecttype* is any valid object type. The expression is **True** if *ObjectName* is of the object type specified by *Objecttype*; otherwise it is **False**.
Statements	One or more statements separated by colons; executed if *Condition* is **True**.
Condition-n	Same as *Condition*.
ElseIfStatements	One or more statements executed if the associated *Condition-n* is **True**.
ElseStatements	One or more statements executed if no previous *Condition* or *Condition-n* expression is **True**.

Remarks

You can use the single-line form (first syntax) for short, simple tests. However, the block form (second syntax) provides more structure and flexibility than the single-line form and is usually easier to read, maintain, and debug.

NOTE

With the single-line syntax, it is possible to have multiple statements executed as the result of an **If...Then** decision, but they must all be on the same line and separated by colons, as in the following statement.

```
If A > 10 Then A = A + 1 : B = B + A : C = C + B
```

When executing a block **If** (second syntax), *Condition* is tested. If *Condition* is **True**, the statements following **Then** are executed. If *Condition* is **False**, each **ElseIf** (if any) is evaluated in turn. When a **True** condition is found, the statements following the associated **Then** are executed. If none of the **ElseIf** statements are **True** (or there are no **ElseIf** clauses), the statements following **Else** are executed. After executing the statements following **Then** or **Else**, execution continues with the statement following **End If**.

The **Else** and **ElseIf** clauses are both optional. You can have as many **ElseIf** statements as you want in a block **If**, but none can appear after the **Else** clause. Block **If** statements can be nested; that is, contained within one another.

What follows the **Then** keyword is examined to determine whether or not a statement is a block **If**. If anything other than a comment appears after **Then** on the same line, the statement is treated as a single-line **If** statement.

A block **If** statement must be the first statement on a line. The block **If** must end with an **End If** statement.

VBScript Statements

2.10.12 On Error Statement

Enables error-handling.

Syntax

On Error Resume Next

Remarks

If you don't use an **On Error Resume Next** statement, any run-time error that occurs is fatal; that is, an error message is displayed and execution stops.

On Error Resume Next causes execution to continue with the statement immediately following the statement that caused the run-time error. The procedure call is considered one statement. This means that if an error occurs inside called procedure (and this procedure doesn't contain the **On Error Resume Next** statement), statements that follow error inside the procedure are not executed, but the script continues with the statement immediately following the procedure call.

This allows execution to continue despite a run-time error. You can then build the error-handling routine inline within the procedure. An **On Error Resume Next** statement becomes inactive when another procedure is called, so you should execute an **On Error Resume Next** statement in each called routine if you want inline error handling within that routine.

VBScript Statements

Example

The following example illustrates use of the **On Error Resume Next** statement.

```
On Error Resume Next
Err.Raise 6 ' Raise an overflow error.
MsgBox ("Error # " & CStr(Err.Number) & " " & Err.Description)
Err.Clear   ' Clear the error.
```

2.10.13 Option Explicit Statement

Forces explicit declaration of all variables in a script.

Syntax

```
Option Explicit
```

Remarks

If used, the **Option Explicit** statement must appear in a script before any other statements.

When you use the **Option Explicit** statement, you must explicitly declare all variables using the Dim, Private, Public, or ReDim statements. If you attempt to use an undeclared variable name, an error occurs.

TIP

Use **Option Explicit** to avoid incorrectly typing the name of an existing variable or to avoid confusion in code where the scope of the variable is not clear.

VBScript Statements

Example

The following example illustrates use of the **Option Explicit** statement.

```
Option Explicit ' Force explicit variable declaration.
Dim MyVar       ' Declare variable.
MyInt = 10      ' Undeclared variable generates error.
MyVar = 10      ' Declared variable does not generate error.
```

2.10.14 Private Statement

Declares private variables and allocates storage space. Declares, in a Class block, a private variable.

Syntax

```
Private VarName[([Subscripts])][, VarName[([Subscripts])]]...
```

The **Private** statement syntax has these parts:

Part	Description
VarName	Name of the variable; follows standard variable naming conventions.
Subscripts	Dimensions of an array variable; up to 60 multiple dimensions may be declared. The *Subscripts* argument uses the following syntax: *upper[, upper]...* The lower bound of an array is always zero.

Remarks

A variable declared with the **Private** statement at a script level (globally) is available in all scripts within the same thread (similar to the **Public** statement). The **Private** statement can still be meaningfully used in a declaration of objects (with the **Class** statement).

A variable that refers to an object must be assigned an existing object using the Set statement before it can be used. Until it is assigned an object, the declared object variable is initialized as **Empty**.

You can also use the **Private** statement with empty parentheses to declare a dynamic array. After declaring a dynamic array, use the ReDim statement within a procedure to define the number of dimensions and elements in the array. If you try to redeclare a dimension for an array variable whose size was explicitly specified in a **Private**, Public, or Dim statement, an error occurs.

 Array Functions

VBScript Statements

Example

The following example illustrates use of the **Private** statement.

```
Private MyNumber   ' Private Variant variable.
Private MyArray(9) ' Private array variable.
' Multiple Private declarations of Variant variables.
Private MyNumber, MyVar, YourNumber
```

2.10.15 Public Statement

Declares public variables and allocates storage space. Declares, in a Class block, a private variable.

Syntax

Public *VarName*[**(**[*Subscripts*]**)**][**,** *VarName*[**(**[*Subscripts*]**)**]]...

The **Public** statement syntax has these parts:

Part	Description
VarName	Name of the variable; follows standard variable naming conventions.
Subscripts	Dimensions of an array variable; up to 60 multiple dimensions may be declared. The *Subscripts* argument uses the following syntax: `upper[, upper]...` The lower bound of an array is always zero.

Remarks

Public statement variables are available to all procedures in all scripts.

A variable that refers to an object must be assigned an existing object using the Set statement before it can be used. Until it is assigned an object, the declared object variable is initialized as **Empty**.

You can also use the **Public** statement with empty parentheses to declare a dynamic array. After declaring a dynamic array, use the ReDim statement within a procedure to define the number of dimensions and elements in the array. If you try to redeclare a dimension for an array variable whose size was explicitly specified in a Private, **Public**, or Dim statement, an error occurs.

Array Functions

VBScript Statements

Example

The following example illustrates the use of the **Public** statement.

```
Public MyNumber   ' Public Variant variable.
Public MyArray(9) ' Public array variable.
' Multiple Public declarations of Variant variables.
Public MyNumber, MyVar, YourNumber
```

2.10.16 Randomize Statement

Initializes the random-number generator.

Syntax

Randomize [*Number*]

The *Number* argument can be any valid numeric expression.

Remarks

Randomize uses *Number* to initialize the Rnd function's random-number generator, giving it a new seed value. If you omit *Number,* the value returned by the system timer is used as the new seed value.

If **Randomize** is not used, the Rnd function (with no arguments) uses the same number as a seed the first time it is called, and thereafter uses the last generated number as a seed value.

NOTE

To repeat sequences of random numbers, call Rnd with a negative argument immediately before using **Randomize** with a numeric argument. Using **Randomize** with the same value for *Number* does not repeat the previous sequence.

VBScript Statements

Example

The following example illustrates use of the **Randomize** statement.

```
Dim MyValue, Response
Randomize                       ' Initialize random-number generator.
Do Until Response = vbNo
   MyValue = Int(( 6 * Rnd) + 1) ' Generate random value between 1 and 6.
   MsgBox MyValue
   Response = MsgBox ("Roll again? ", vbYesNo)
Loop
```

2.10.17 ReDim Statement

Declares dynamic-array variables, and allocates or reallocates storage space at procedure level.

Syntax

ReDim *[***Preserve***] VarName***(***Subscripts***)***[***, VarName***(***Subscripts***)***]...***

The **ReDim** statement syntax has these parts:

Part	Description
Preserve	Preserves the data in an existing array when you change the size of the last dimension.
VarName	Name of the variable; follows standard variable naming conventions.

Subscripts	Dimensions of an array variable; up to 60 multiple dimensions may be declared. The *Subscripts* argument uses the following syntax: *upper[, upper]...* The lower bound of an array is always zero.

Remarks

The **ReDim** statement is used to size or resize a dynamic array that has already been formally declared using a Private, Public, or Dim statement with empty parentheses (without dimension subscripts). You can use the **ReDim** statement repeatedly to change the number of elements and dimensions in an array.

If you use the **Preserve** keyword, you can resize only the last array dimension, and you can't change the number of dimensions at all. For example, if your array has only one dimension, you can resize that dimension because it is the last and only dimension. However, if your array has two or more dimensions, you can change the size of only the last dimension and still preserve the contents of the array.

The following example shows how you can increase the size of the last dimension of a dynamic array without erasing any existing data contained in the array.

```
ReDim X(10, 10, 10)
  '...
ReDim Preserve X(10, 10, 15)
```

CAUTION

If you make an array smaller than it was originally, data in the eliminated elements is lost.

When variables are initialized, a numeric variable is initialized to 0 and a string variable is initialized to a zero-length string (""). A variable that refers to an object must be assigned an existing object using the Set statement before it can be used. Until it is assigned an object, the declared object variable has the special value **Nothing**.

Array Functions

VBScript Statements

2.10.18 Rem Statement

Includes explanatory remarks in a program.

Syntax

Rem *Comment*

or

' Comment

The *Comment* argument is the text of any comment you want to include. After the **Rem** keyword, a space is required before *Comment*.

Remarks

As shown in the syntax section, you can use an apostrophe (') instead of the **Rem** keyword. If the **Rem** keyword follows other statements on a line, it must be separated from the statements by a colon. However, when you use an apostrophe, the colon is not required after other statements.

VBScript Statements

Example

The following example illustrates the use of the **Rem** statement.

```
Dim MyStr1, MyStr2
MyStr1 = "Hello" : Rem Comment after a statement separated by a colon.
MyStr2 = "Goodbye"     ' This is also a comment; no colon is needed.
Rem Comment on a line with no code; no colon is needed.
```

2.10.19 Select Case Statement

Executes one of several groups of statements, depending on the value of an expression.

Syntax

Select Case *TestExpression*

*[***Case** *ExpressionList-n*

[Statements-n]]...

*[***Case Else** *ExpressionList-n*

[ElseStatements-n]]

End Select

The **Select Case** statement syntax has these parts:

Part	Description
TestExpression	Any numeric or string expression.
ExpressionList-n	Required if **Case** appears. Delimited list of one or more expressions.
Statements-n	One or more statements executed if *TestExpression* matches any part of *ExpressionList-n*.
ElseStatements-n	One or more statements executed if *TestExpression* doesn't match any of the **Case** clauses.

Remarks

If *TestExpression* matches any **Case** *ExpressionList* expression, the statements following that **Case** clause are executed up to the next **Case** clause, or for the last clause, up to **End Select**. Control then passes to the statement following **End Select**. If *TestExpression* matches an *ExpressionList* expression in more than one **Case** clause, only the statements following the first match are executed.

The **Case Else** clause is used to indicate the *ElseStatements* to be executed if no match is found between the *TestExpression* and an *ExpressionList* in any of the other **Case** selections. Although not required, it is a good idea to have a **Case Else** statement in your **Select Case** block to handle unforeseen *TestExpression* values. If no **Case** *ExpressionList* matches *TestExpression* and there is no **Case Else** statement, execution continues at the statement following **End Select**.

Select Case statements can be nested. Each nested **Select Case** statement must have a matching **End Select** statement.

VBScript Statements

Example

The following example illustrates the use of the **Select Case** statement.

```
Dim Color, MyVar
Sub ChangeBackground (Color)
  MyVar = lcase (Color)
  Select Case MyVar
    Case "red"
      document.bgColor = "red"
    Case "green"
      document.bgColor = "green"
    Case "blue"
      document.bgColor = "blue"
    Case Else
      MsgBox "pick another color"
  End Select
End Sub
```

2.10.20 Set Statement

Assigns an object reference to a variable or property, or associates a procedure reference with an event.

Syntax 1

Set *ObjectVar* = *{ObjectExpression* | **New** *ClassName* | **Nothing***}*

Syntax 2

Set *Object.EventName* = **GetRef***(ProcName)*

The **Set** statement syntax has these parts:

Part	Description
ObjectVar	Required. Name of the variable or property; follows standard variable naming conventions.
ObjectExpression	Optional. Expression consisting of the name of an object, another declared variable of the same object type, or a function or method that returns an object of the same object type.
New	Keyword used to create a new instance of a class. If *ObjectVar* contained a reference to an object, that reference is released when the new one is assigned. The **New** keyword can only be used to create an instance of a class.
ClassName	Optional. Name of the class being created. A class and its members are defined using the Class statement.
Nothing	Optional. Discontinues association of *ObjectVar* with any specific object or class. Assigning *ObjectVar* to **Nothing** releases all the system and memory resources associated with the previously referenced object when no other variable refers to it.
Object	Required. Name of the object with which *EventName* is associated.
EventName	Required. Name of the event to which the function is to be bound.
ProcName	Required. String containing the name of the Sub or Function being associated with the *EventName*.

Remarks

To be valid, *ObjectVar* must be an object type consistent with the object being assigned to it.

The Dim, Private, Public, or ReDim statements only declare a variable that refers to an object. No actual object is referred to until you use the **Set** statement to assign a specific object.

Generally, when you use **Set** to assign an object reference to a variable, no copy of the object is created for that variable. Instead, a reference to the object is created. More than one object variable can refer to the same object. Because these variables are references to (rather than copies of) the object, any change in the object is reflected in all variables that refer to it.

Using the **New** keyword allows you to concurrently create an instance of a class and assign it to an object reference variable. The variable to which the instance of the class is being assigned must already have been declared with the Dim (or equivalent) statement.

Refer to the documentation for the GetRef function for information on using **Set** to associate a procedure with an event.

VBScript Statements

Example

The following example illustrates the use of the **Set** statement.

```
Function ShowFreeSpace(drvPath)
  Dim fso, d, s
  Set fso = CreateObject("Scripting.FileSystemObject")
  Set d = fso.GetDrive(fso.GetDriveName(drvPath))
  s = "Drive " & UCase(drvPath) & " - "
  s = s & d.VolumeName & "<BR>"
  s = s & "Free Space: " & FormatNumber(d.FreeSpace / 1024, 0)
  s = s & " Kbytes"
  ShowFreeSpace = s
End Function
```

2.10.21 Stop Statement

Interrupts the code execution and launches an external tool for debugging scripts (Just-In-Time debugger) if debugging is enabled.

Syntax

Stop

Remarks

A Just-In-Time debugger must be installed (e.g. Microsoft Script Debugger or the tool contained in Microsoft Visual Studio).

Script debugging must be enabled in the Windows operating system. This can be done by activating the *Enable script debugging with external tool (Just-In-Time debugger)* option (Reliance Design > Tools > Environment Options).

VBScript Statements

2.10.22 Sub Statement

Declares the name, arguments, and code that form the body of a **Sub** procedure.

Syntax

[**Public** | **Private**] **Sub** *Name [***(***ArgList***)***]*

[Statements]

*[***Exit Sub***]*

[Statements]

End Sub

The **Sub** statement syntax has these parts:

Part	Description

Public	Indicates that the **Sub** procedure is accessible to all other procedures in all scripts.
Private	Indicates that the **Sub** procedure is accessible only to other procedures in the script where it is declared.
Name	Name of the **Sub**; follows standard variable naming conventions.
ArgList	List of variables representing arguments that are passed to the **Sub** procedure when it is called. Multiple variables are separated by commas.
Statements	Any group of statements to be executed within the body of the **Sub** procedure.

The *ArgList* argument has the following syntax and parts:

[**ByVal** | **ByRef**] *VarName*[**( )**]

Part	Description
ByVal	Indicates that the argument is passed by value.
ByRef	Indicates that the argument is passed by reference.
VarName	Name of the variable representing the argument; follows standard variable naming conventions.

Remarks

If not explicitly specified using either Public or Private, **Sub** procedures are public by default, that is, they are visible to all other procedures in your script. The value of local variables in a **Sub** procedure is not preserved between calls to the procedure.

You can't define a **Sub** procedure inside any other procedure (e.g. Function or Property Get).

The Exit **Sub** statement causes an immediate exit from a **Sub** procedure. Program execution continues with the statement that follows the statement that called the **Sub** procedure. Any number of Exit **Sub** statements can appear anywhere in a **Sub** procedure.

Like a Function procedure, a **Sub** procedure is a separate procedure that can take arguments, perform a series of statements, and change the value of its arguments. However, unlike a Function procedure, which returns a value, a **Sub** procedure can't be used in an expression.

You call a **Sub** procedure using the procedure name followed by the argument list. See the Call statement for specific information on how to call **Sub** procedures.

CAUTION

Sub procedures can be recursive, that is, they can call themselves to perform a given task. However, recursion can lead to stack overflow.

Variables used in **Sub** procedures fall into two categories: those that are explicitly declared within the procedure and those that are not. Variables that are explicitly declared in a procedure (using Dim or the equivalent) are always local to the procedure. Variables that are used but not explicitly declared in a procedure are also local, unless they are explicitly declared at some higher level outside the procedure.

CAUTION

A procedure can use a variable that is not explicitly declared in the procedure, but a naming conflict can occur if anything you have defined at the script level has the same name. If your procedure refers to an undeclared variable that has the same name as another procedure, constant or variable, it is assumed that your procedure is referring to that script-level name. To avoid this kind of conflict, use an Option Explicit statement to force explicit declaration of variables.

VBScript Statements

2.10.23 While...WEnd Statement

Executes a series of statements as long as a given condition is **True**.

Syntax

While *Condition*

[Statements]

WEnd

The **While...WEnd** statement syntax has these parts:

Part	Description
Condition	Numeric or string expression that evaluates to **True** or **False**. If *Condition* is **Null**, *Condition* is treated as **False**.
Statements	One or more statements executed while condition is **True**.

Remarks

If *Condition* is **True**, all statements in *Statements* are executed until the **WEnd** statement is encountered. Control then returns to the **While** statement and *Condition* is again checked. If *Condition* is still **True**, the process is repeated. If it is not **True**, execution resumes with the statement following the **WEnd** statement.

While...WEnd loops can be nested to any level. Each **WEnd** matches the most recent **While**.

TIP

The Do...Loop statement provides a more structured and flexible way to perform looping.

VBScript Statements

Example

The following example illustrates use of the **While...WEnd** statement.

```
Dim Counter
Counter = 0              ' Initialize variable.
While Counter < 20       ' Test value of Counter.
  Counter = Counter + 1 ' Increment Counter.
```

```
    Alert Counter
WEnd                    ' End While loop when Counter > 19.
```

2.10.24 With Statement

Executes a series of statements on a single object.

Syntax

With *Object*

Statements

End With

The **With** statement syntax has these parts:

Part	Description
Object	Required. Name of an object or a function that returns an object.
Statements	Required. One or more statements to be executed on *Object*.

Remarks

The **With** statement allows you to perform a series of statements on a specified object without requalifying the name of the object. For example, to change a number of different properties on a single object, place the property assignment statements within the **With** control structure, referring to the object once instead of referring to it with each property assignment. The following example illustrates use of the **With** statement to assign values to several properties of the same object.

```
With MyLabel
  .Height = 2000
  .Width = 2000
  .Caption = "This is MyLabel"
End With
```

While property manipulation is an important aspect of **With** functionality, it is not the only use. Any legal code can be used within a **With** block.

NOTE

Once a **With** block is entered, *Object* can't be changed. As a result, you can't use a single **With** statement to affect a number of different objects.

You can nest **With** statements by placing one **With** block within another. However, because members of outer **With** blocks are masked within the inner **With** blocks, you must provide a fully qualified object reference in an inner **With** block to any member of an object in an outer **With** block.

IMPORTANT

Do not jump into or out of **With** blocks. If statements in a **With** block are executed, but either the **With** or **End With** statement is not executed, you may get errors or unpredictable behavior.

VBScript Statements

2.11 VBScript Constants

VBScript definuje konstanty pro zjednodušení programování. Následující konstanty mohou být použity kdekoli v kódu místo skutečných hodnot:

- **Color** Constants
- **Comparison** Constants
- **Date and Time** Constants
- **Date Format** Constants
- **Miscellaneous** Constants
- **MsgBox** Constants
- **String** Constants
- **Tristate** Constants
- **VarType** Constants

2.11.1 Color Constants

Since these constants are built into *VBScript*, you don't have to define them before using them. Use them anywhere in your code to represent the values shown for each.

Constant	Value	Description
vbBlack	&h00	Black
vbRed	&hFF	Red
vbGreen	&hFF00	Green
vbYellow	&hFFFF	Yellow
vbBlue	&hFF0000	Blue
vbMagenta	&hFF00FF	Magenta
vbCyan	&hFFFF00	Cyan
vbWhite	&hFFFFFF	White

VBScript Constants

2.11.2 Comparison Constants

Since these constants are built into *VBScript*, you don't have to define them before using them. Use them anywhere in your code to represent the values shown for each.

Constant	Value	Description
vbBinaryCompare	0	Perform a binary comparison.
vbTextCompare	1	Perform a textual comparison.

VBScript Constants

2.11.3 Date and Time Constants

Since these constants are built into *VBScript*, you don't have to define them before using them. Use them anywhere in your code to represent the values shown for each.

Constant	Value	Description
vbSunday	1	Sunday
vbMonday	2	Monday
vbTuesday	3	Tuesday
vbWednesday	4	Wednesday
vbThursday	5	Thursday
vbFriday	6	Friday
vbSaturday	7	Saturday
vbUseSystem	0	Use the date format contained in the regional settings for your computer.
vbUseSystemDayOfWeek	0	Use the day of the week specified in your system settings for the first day of the week.

vbFirstJan1	1	Use the week in which January 1 occurs (default).
vbFirstFourDays	2	Use the first week that has at least four days in the new year.
vbFirstFullWeek	3	Use the first full week of the year.

VBScript Constants

2.11.4 Date Format Constants

Since these constants are built into *VBScript*, you don't have to define them before using them. Use them anywhere in your code to represent the values shown for each.

Constant	Value	Description
vbGeneralDate	0	Display a date and/or time. For real numbers, display a date and time. If there is no fractional part, display only a date. If there is no integer part, display time only. Date and time display is determined by your system settings.
vbLongDate	1	Display a date using the long date format specified in your computer's regional settings.
vbShortDate	2	Display a date using the short date format specified in your computer's regional settings.
vbLongTime	3	Display a time using the long time format specified in your computer's regional settings.
vbShortTime	4	Display a time using the short time format specified in your computer's regional settings.

VBScript Constants

2.11.5 Miscellaneous Constants

Since this constant is built into *VBScript*, you don't have to define it before using it. Use it anywhere in your code to represent the values shown.

Constant	Value	Description
vbObjectError	-2147221504	User-defined error numbers should be greater than this value.

VBScript Constants

Example

```
Err.Raise Number = vbObjectError + 1000
```

2.11.6 MsgBox Constants

The following constants are used with the MsgBox function to identify what buttons and icons appear on a message box and which button is the default. In addition, the modality of the MsgBox can be specified. Since these constants are built into *VBScript*, you don't have to define them before using them. Use them anywhere in your code to represent the values shown for each.

Constant	Value	Description
vbOKOnly	0	Display *OK* button only.
vbOKCancel	1	Display *OK* and *Cancel* buttons.
vbAbortRetryIgnore	2	Display *Abort*, *Retry*, and *Ignore* buttons.
vbYesNoCancel	3	Display *Yes*, *No*, and *Cancel* buttons.
vbYesNo	4	Display *Yes* and *No* buttons.

vbRetryCancel	5	Display *Retry* and *Cancel* buttons.
vbCritical	16	Display *Critical Message* icon.
vbQuestion	32	Display *Warning Query* icon.
vbExclamation	48	Display *Warning Message* icon.
vbInformation	64	Display *Information Message* icon.
vbDefaultButton1	0	First button is default.
vbDefaultButton2	256	Second button is default.
vbDefaultButton3	512	Third button is default.
vbDefaultButton4	768	Fourth button is default.
vbApplicationModal	0	Application modal; the user must respond to the message box before continuing work in the current application.
vbSystemModal	4096	System modal; all applications are suspended until the user responds to the message box.

The following constants are used with the MsgBox function to identify which button a user has selected. These constants are only available when your project has an explicit reference to the appropriate type library containing these constant definitions. For *VBScript*, you must explicitly declare these constants in your code.

Constant	Value	Description
vbOK	1	*OK* button was clicked.
vbCancel	2	*Cancel* button was clicked.
vbAbort	3	*Abort* button was clicked.

vbRetry	4	*Retry* button was clicked.
vbIgnore	5	*Ignore* button was clicked.
vbYes	6	*Yes* button was clicked.
vbNo	7	*No* button was clicked.

VBScript Constants

2.11.7 String Constants

Since these constants are built into *VBScript*, you don't have to define them before using them. Use them anywhere in your code to represent the values shown for each.

Constant	Value	Description
vbCr	**Chr(**13**)**	Carriage return.
vbCrLf	**Chr(**13**) & Chr(**10**)**	Carriage return–linefeed combination.
vbFormFeed	**Chr(**12**)**	Form feed; not useful in Microsoft Windows.
vbLf	**Chr(**10**)**	Line feed.
vbNewLine	**Chr(**13**) & Chr(**10**)** or **Chr(**10**)**	Platform-specific newline character; whatever is appropriate for the platform.
vbNullChar	**Chr(**0**)**	Character having the value 0.
vbNullString	String having value 0	Not the same as a zero-length string (""); used for calling external procedures.
vbTab	**Chr(**9**)**	Horizontal tab.
vbVerticalTab	**Chr(**11**)**	Vertical tab; not useful in Microsoft Windows.

VBScript Constants

2.11.8 Tristate Constants

Since these constants are built into *VBScript*, you don't have to define them before using them. Use them anywhere in your code to represent the values shown for each.

Constant	Value	Description
vbUseDefault	-2	Use default from computer's regional settings.
vbTrue	-1	**True**
vbFalse	0	**False**

 VBScript Constants

2.11.9 VarType Constants

These constants are only available when your project has an explicit reference to the appropriate type library containing these constant definitions. For *VBScript*, you must explicitly declare these constants in your code.

Constant	Value	Description
vbEmpty	0	Uninitialized (default).
vbNull	1	Contains no valid data.
vbInteger	2	Integer subtype.
vbLong	3	Long subtype.
vbSingle	4	Single subtype.
vbSingle	5	Double subtype.
vbCurrency	6	Currency subtype.
vbDate	7	Date subtype.
vbString	8	String subtype.

vbObject	9	Object.
vbError	10	Error subtype.
vbBoolean	11	Boolean subtype.
vbVariant	12	Variant (used only for arrays of variants).
vbDataObject	13	Data access object.
vbDecimal	14	Decimal subtype.
vbByte	17	Byte subtype.
vbArray	8192	Array.


VBScript Constants

2.12 VBScript Operators

- **Addition** Operator (+)
- **And** Operator
- **Assignment** Operator (=)
- **Concatenation** Operator (&)
- **Division** Operator (/)
- **Eqv** Operator
- **Exponentiation** Operator (^)
- **Imp** Operator
- **Integer** Division Operator (\)
- **Is** Operator
- **Mod** Operator
- **Multiplication** Operator (*)
- **Not** Operator
- **Or** Operator
- **Subtraction** Operator (-)
- **Xor** Operator

2.12.1 Addition Operator (+)

Sums two numbers.

Syntax

Result = *Expression1* + *Expression2*

The **+** operator syntax has these parts:

Part	Description
Result	Any numeric variable.
Expression1	Any expression.

Expression2	Any expression.

Remarks

Although you can also use the **+** operator to concatenate two character strings, you should use the & operator for concatenation to eliminate ambiguity and provide self-documenting code.

When you use the **+** operator, you may not be able to determine whether addition or string concatenation will occur.

The underlying subtype of the expressions determines the behavior of the **+** operator in the following way:

If	Then
Both expressions are numeric	Add.
Both expressions are strings	Concatenate.
One expression is numeric and the other is a string	Add.

If one or both expressions are **Null** expressions, *Result* is **Null**. If both expressions are **Empty**, *Result* is an **Integer** subtype. However, if only one expression is **Empty**, the other expression is returned unchanged as *Result*.

VBScript Operators

2.12.2 And Operator

Performs a logical conjunction on two expressions.

Syntax

Result = *Expression1* **And** *Expression2*

The **And** operator syntax has these parts:

Part	Description
Result	Any numeric variable.

Expression1	Any expression.
Expression2	Any expression.

Remarks

If, and only if, both expressions evaluate to **True**, *Result* is **True**. If either expression evaluates to **False**, *Result* is **False**. The following table illustrates how *Result* is determined:

If **Expression1** *is*	*And* **Expression2** *is*	*The* **Result** *is*
True	**True**	**True**
True	**False**	**False**
True	**Null**	**Null**
False	**True**	**False**
False	**False**	**False**
False	**Null**	**False**
Null	**True**	**Null**
Null	**False**	**False**
Null	**Null**	**Null**

The **And** operator also performs a bitwise comparison of identically positioned bits in two numeric expressions and sets the corresponding bit in *Result* according to the following table:

If bit in **Expression1** *is*	*And bit in* **Expression2** *is*	*The* **Result** *is*
0	0	0
0	1	0
1	0	0
1	1	1

VBScript Operators

2.12.3 Assignment Operator

Assigns a value to a variable or property.

Syntax

Variable = Value

The **=** operator syntax has these parts:

Part	Description
Variable	Any variable or any writable property.
Value	Any numeric or string literal, constant, or expression.

Remarks

The name on the left side of the equal sign can be a simple scalar variable or an element of an array. Properties on the left side of the equal sign can only be those properties that are writable at run time.

VBScript Operators

2.12.4 Concatenation Operator (&)

Forces string concatenation of two expressions.

Syntax

Result = Expression1 & Expression2

The **&** operator syntax has these parts:

Part	Description
Result	Any variable.
Expression1	Any expression.

Expression2	Any expression.

Remarks

Whenever an *Expression* is not a string, it is converted to a **String** subtype. If both expressions are **Null**, *Result* is also **Null**. However, if only one *Expression* is **Null**, that expression is treated as a zero-length string ("") when concatenated with the other expression. Any expression that is **Empty** is also treated as a zero-length string.

VBScript Operators

2.12.5 Division Operator (/)

Divides two numbers and returns a floating-point result.

Syntax

Result = Number1 / Number2

The / operator syntax has these parts:

Part	Description
Result	Any numeric variable.
Number1	Any numeric expression.
Number2	Any numeric expression.

Remarks

If one or both expressions are **Null** expressions, *Result* is **Null**. Any expression that is **Empty** is treated as 0.

VBScript Operators

2.12.6 Eqv Operator

Performs a logical equivalence on two expressions.

Syntax

Result = *Expression1* **Eqv** *Expression2*

The **Eqv** operator syntax has these parts:

Part	Description
Result	Any numeric variable.
Expression1	Any expression.
Expression2	Any expression.

Remarks

If either expression is **Null**, *Result* is also **Null**. When neither expression is **Null**, *Result* is determined according to the following table:

If **Expression1** *is*	*And* **Expression2** *is*	*The* **Result** *is*
True	**True**	**True**
True	**False**	**False**
False	**True**	**False**
False	**False**	**True**

The **Eqv** operator performs a bitwise comparison of identically positioned bits in two numeric expressions and sets the corresponding bit in *Result* according to the following table:

If bit in **Expression1** *is*	*And bit in* **Expression2** *is*	*The* **Result** *is*
0	0	1
0	1	0
1	0	0
1	1	1

VBScript Operators

2.12.7 Exponentiation Operator (^)

Raises a number to the power of an exponent.

Syntax

Result = *Number* ^ *Exponent*

The **^** operator syntax has these parts:

Part	Description
Result	Any numeric variable.
Number	Any numeric expression.
Exponent	Any numeric expression.

Remarks

Number can be negative only if *Exponent* is an integer value. When more than one exponentiation is performed in a single expression, the **^** operator is evaluated as it is encountered from left to right.

If either *Number* or *Exponent* is a **Null** expression, *Result* is also **Null**.


VBScript Operators

2.12.8 Imp Operator

Performs a logical implication on two expressions.

Syntax

Result = *Expression1* **Imp** *Expression2*

The **Imp** operator syntax has these parts:

Part	Description
Result	Any numeric variable.
Expression1	Any expression.

Expression2	Any expression.

Remarks

The following table illustrates how *Result* is determined:

If **Expression1** *is*	*And* **Expression2** *is*	*The* **Result** *is*
True	**True**	**True**
True	**False**	**False**
True	**Null**	**Null**
False	**True**	**True**
False	**False**	**True**
False	**Null**	**True**
Null	**True**	**True**
Null	**False**	**Null**
Null	**Null**	**Null**

The **Imp** operator performs a bitwise comparison of identically positioned bits in two numeric expressions and sets the corresponding bit in *Result* according to the following table:

If bit in **Expression1** *is*	*And bit in* **Expression2** *is*	*The* **Result** *is*
0	0	1
0	1	1
1	0	0
1	1	1

VBScript Operators

2.12.9 Integer Division Operator (\)

Divides two numbers and returns an integer result.

Syntax

Result = *Number1* \ *Number2*

The \ operator syntax has these parts:

Part	Description
Result	Any numeric variable.
Number1	Any numeric expression.
Number2	Any numeric expression.

Remarks

Before division is performed, numeric expressions are rounded to **Byte**, **Integer**, or **Long** subtype expressions.

If any expression is **Null**, *Result* is also **Null**. Any expression that is **Empty** is treated as 0.

 VBScript Operators

2.12.10 Is Operator

Compares two object reference variables.

Syntax

Result = *Object1* **Is** *Object2*

The **Is** operator syntax has these parts:

Part	Description
Result	Any numeric variable.
Object1	Any object name.
Object2	Any object name.

Remarks

If *Object1* and *Object2* both refer to the same object, *Result* is **True**; if they do not, *Result* is **False**. Two variables can be made to refer to the same object in several ways.

In the following example, A has been set to refer to the same object as B.

```
Set A = B
```

The following example makes A and B refer to the same object as C.

```
Set A = C

Set B = C
```

VBScript Operators

2.12.11 Mod Operator

Divides two numbers and returns only the remainder.

Syntax

Result = *Number1* **Mod** *Number2*

The **Mod** operator syntax has these parts:

Part	Description
Result	Any numeric variable.
Number1	Any numeric expression.
Number2	Any numeric expression.

Remarks

The modulus, or remainder, operator divides *Number1* by *Number2* (rounding floating-point numbers to integers) and returns only the remainder as *Result*. For example, in the following expression, A (which is *Result*) equals 5.

```
A = 19 Mod 6.7
```

If any expression is **Null**, *Result* is also **Null**. Any expression that is **Empty** is treated as 0.

VBScript Operators

2.12.12 Multiplication Operator (*)

Multiplies two numbers.

Syntax

Result = *Number1* ***** *Number2*

The * operator syntax has these parts:

Part	Description
Result	Any numeric variable.
Number1	Any numeric expression.
Number2	Any numeric expression.

Remarks

If one or both expressions are **Null** expressions, *Result* is **Null**. If an expression is **Empty**, it is treated as if it were 0.

VBScript Operators

2.12.13 Negation Operator (-)

Finds the difference between two numbers or indicates the negative value of a numeric expression.

Syntax 1

Result = *Number1* - *Number2*

Syntax 2

-*Number*

The **-** operator syntax has these parts:

Part	Description
Result	Any numeric variable.
Number	Any numeric expression.
Number1	Any numeric expression.
Number2	Any numeric expression.

Remarks

In Syntax 1, the - operator is the arithmetic subtraction operator used to find the difference between two numbers. In Syntax 2, the - operator is used as the unary negation operator to indicate the negative value of an expression.

If one or both expressions are **Null** expressions, *Result* is **Null**. If an expression is **Empty**, it is treated as if it were 0.

VBScript Operators

2.12.14 Not Operator

Performs logical negation on an expression.

Syntax

Result = **Not** *Expression*

The **Not** operator syntax has these parts:

Part	Description
Result	Any numeric variable.
Expression	Any expression.

Remarks

The following table illustrates how *Result* is determined:

If **Expression1** *is*	*The* **Result** *is*

True	**False**
False	**True**
Null	**Null**

In addition, the **Not** operator inverts the bit values of any variable and sets the corresponding bit in *Result* according to the following table:

Bit in **Expression**	*Bit in* **Result**
0	1
1	0

VBScript Operators

2.12.15 Or Operator

Performs a logical disjunction on two expressions.

Syntax

Result = *Expression1* **Or** *Expression2*

The **Or** operator syntax has these parts:

Part	**Description**
Result	Any numeric variable.
Expression1	Any expression.
Expression2	Any expression.

Remarks

If either or both expressions evaluate to **True**, *Result* is **True**. The following table illustrates how *Result* is determined:

If **Expression1** *is*	*And* **Expression2** *is*	*The* **Result** *is*

True	**True**	**True**
True	**False**	**True**
True	**Null**	**True**
False	**True**	**True**
False	**False**	**False**
False	**Null**	**Null**
Null	**True**	**True**
Null	**False**	**Null**
Null	**Null**	**Null**

The **Or** operator also performs a bitwise comparison of identically positioned bits in two numeric expressions and sets the corresponding bit in *Result* according to the following table:

If bit in **Expression1** *is*	*And bit in* **Expression2** *is*	*The* **Result** *is*
0	0	0
0	1	1
1	0	1
1	1	1

VBScript Operators

2.12.16 Subtraction Operator (-)

Finds the difference between two numbers or indicates the negative value of a numeric expression.

Syntax 1

Result = Number1 - Number2

Syntax 2

-Number

The **-** operator syntax has these parts:

Part	Description
Result	Any numeric variable.
Number	Any numeric expression.
Number1	Any numeric expression.
Number2	Any numeric expression.

Remarks

In Syntax 1, the - operator is the arithmetic subtraction operator used to find the difference between two numbers. In Syntax 2, the - operator is used as the unary negation operator to indicate the negative value of an expression.

If one or both expressions are **Null** expressions, *Result* is **Null**. If an expression is **Empty**, it is treated as if it were 0.

VBScript Operators

2.12.17 Xor Operator

Performs a logical exclusion on two expressions.

Syntax

Result = *Expression1* **Xor** *Expression2*

The **Xor** operator syntax has these parts:

Part	Description
Result	Any numeric variable.
Expression1	Any expression.
Expression2	Any expression.

Remarks

If one, and only one, of the expressions evaluates to **True**, *Result* is **True**. However, if either expression is **Null**, *Result* is also **Null**. When neither expression is **Null**, *Result* is determined according to the following table:

If **Expression1** *is*	*And* **Expression2** *is*	*The* **Result** *is*
True	**True**	**False**
True	**False**	**True**
False	**True**	**True**
False	**False**	**False**

The **Xor** operator also performs a bitwise comparison of identically positioned bits in two numeric expressions and sets the corresponding bit in *Result* according to the following table:

If bit in **Expression1** *is*	*And bit in* **Expression2** *is*	*The* **Result** *is*
0	0	0
0	1	1
1	0	1
1	1	0

VBScript Operators

3 Reliance-defined Objects

3.1 Reliance-defined Objects

Reliance expands *VBScript* by several objects designed for accessing the runtime environment.

Each object implements methods and/or properties intended for a specific set of operations:

- **RAlm** Object – for operations on alarms defined in a visualization project
- **RDb** Object – for operations on databases defined in a visualization project
- **RDev** Object – for operations on devices defined in a visualization project
- **RError** Object – for accessing information on the result of any of the above operations
- **RInet** Object – for E-mail operations
- **RModem** Object – for GSM modem operations
- **RScr** Object – for operations on scripts defined in a visualization project
- **RSys** Object – for miscellaneous operations
- **TTable**-type Objects – for operations on database tables, either current or archive, belonging to databases defined in a visualization project
- **RTag** Object – for operations on tags defined in a visualization project
- **RUser** Object – for operations on users defined in a visualization project
- **RWS** Object – for accessing the Web service of Reliance data servers

These objects, with the exception of **TTable**-type objects, are created and initialized automatically during starting a visualization project. When using these objects, follow the rules for working with properties and methods of objects.

3.2 Execution of Scripts in the Runtime Environment

Scripts to be executed are placed in a script queue to a position that depends on their priority. The scripts queued for execution are sequentially processed in the background without affecting the user interface of the runtime environment. If a syntax error occurs in a script, execution of the script is terminated. If an error other than syntax occurs in a script, execution of the script is terminated based on the *Terminate script on error* option (*Reliance Design* > *Project* > *Options* > *Scripts* > *Other*). Information on any error that may occur when accessing properties and calling methods of Reliance-defined objects is stored in the RError object and can be accessed through its properties after every such property access or method call.

3.3 Processing of Data Passed to Scripts from the Runtime Environment

A parameter can be passed to a script, for example, when running the script by clicking a visual component. Using the value of the parameter in the script's code it is possible to determine which event (e.g. which component was clicked) triggered the script. A script can retrieve the values of parameters by reading properties of a special data object returned by a call to the RScr.GetCurrentScriptData or RScr.GetCurrentScriptDataEx in the script's code.

3.4 Working with Global Constants, Variables, Procedures and Functions

A script containing the declaration of a global constant, variable, procedure or function must be executed before any other script that references the constant or variable, or calls the procedure or function.

TIP

It is advisable to place the declarations of global constants, variables, procedures or functions to a single script and activate its *Run on thread initialization* property (*Reliance Design* > *Managers* > *Script Manager* > script properties > the *Advanced* page). This way, it is guaranteed that the script will execute before other scripts.

3.5 Tips for Writing Scripts

To prevent errors due to typing mistakes, it is recommended to use the Option Explicit statement in all scripts. Thus, if a script references an undeclared variable (e.g. due to a typing mistake), a syntax error occurs.

The MsgBox and InputBox functions should only be used for debugging purposes. Each of these functions displays a dialog box and stops execution of the script from which it has been called. The script does not continue execution until the dialog box is closed by the user. No other scripts in the thread can be processed either since only one script can execute per thread. Another reason for avoiding using these functions is that the dialog box may appear in the background of the program where it cannot be seen by the user, thus stopping processing of scripts. When using the MsgBox function, this behavior can be prevented by passing the `vsSystemModal` constant as the second parameter to the function. For example, `MsgBox "Message text", vbSystemModal`. As a result, the message box will stay on top of other windows.

3.6 RAlm Object

The **RAlm** object implements methods for operations on alarms.

Methods:

- **RAlm.AckAlarm** Procedure
- **RAlm.AckAllAlarms** Procedure
- **RAlm.CreateAlarm** Procedure
- **RAlm.CurrentAlarms** Procedure
- **RAlm.CurrentAlarmsByDevice** Procedure
- **RAlm.DbAlarms** Procedure
- **RAlm.DbAlarmsByDevice** Procedure
- **RAlm.DbAlarmsByFilter** Procedure
- **RAlm.DisableDeviceAlarms** Procedure
- **RAlm.EnableDeviceAlarms** Procedure

Other:

- **Alarm Type** Constants
- **Alarm Triggering Condition** Constants

3.6.1 RAlm.AckAlarm Procedure

RAlm.AckAlarm acknowledges an alarm.

Syntax

RAlm.AckAlarm *DevName*, *AlmName*: **String**

Argument	Description
DevName	The name of the device that the alarm belongs to.
AlmName	The name of the alarm.

RAlm Object

Example

```
' Acknowledge the alarm PumpFailure from the device PLC1.
RAlm.AckAlarm "PLC1", "PumpFailure"
```

3.6.2 RAlm.AckAllAlarms Procedure

RAlm.AckAllAlarms acknowledges all unacknowledged alarms.

Syntax

RAlm.AckAllAlarms

RAlm Object

Example

```
If RTag.GetTagValue("System", "AckAlarms") Then
  ' Acknowledge all alarms depending on the value
  ' of the tag AckAlarms from the device System.
  RAlm.AckAllAlarms
End If
```

3.6.3 RAlm.CreateAlarm Procedure

RAlm.CreateAlarm generates an alarm using specified parameters.

Syntax

RAlm.CreateAlarm *Type*: **Byte;** *Text*, *Comment*: **String;** *StartTime*: **DateTime;** *Save*, *Show*, *Print*, *Ack*, *ActivateWnd*: **Bool;** *AckRights*: **Integer;** *StartSound*: **String;** *StartScript*, *AckScript*: **Variant;** *RelatedWnd*: **String**

Argument	Description
AlarmType	Alarm type (Alarm Type Constants).
Text	The text of the alarm.

Comment	The comment for the alarm.
StartTime	Date and time when the alarm was generated.
Save	Determines whether to log the alarm to the alarm database.
Show	Determines whether to display the alarm in the list of current alarms.
Print	Determines whether to print the alarm online.
Ack	Determines whether it is required that the alarm be acknowledged by the user (operator).
ActivateWnd	Determines whether to activate the list of current alarms (makes sense only if *Show* = **True**).
AckRights	The access rights required for acknowledging the alarm (makes sense only if *Ack* = **True**).
StartSound	The file name of a sound to be played after the alarm is generated.
StartScript	The name or ID of a script to be executed after the alarm is generated.
AckScript	The name or ID of a script to be executed after the alarm is acknowledged (makes sense only if *Ack* = **True**).
RelatedWnd	The name of the window related to the alarm.

Remarks

An alarm generated by calling the method is processed in the same way as internal messages (i.e. messages defined not in a visualization project via the *Device Manager* but in the program code of the runtime software). It is not associated with a tag and cannot be transferred to another computer through a network connection.

RAlm Object

Example

```
Dim fso, drv, freesp
' Create the Scripting.FileSystemObject object and assign it to the variable fso.
Set fso = CreateObject("Scripting.FileSystemObject")
' Create an object representing the C: drive and assign it to the variable drv.
Set drv = fso.GetDrive(fso.GetDriveName("C:\"))
freesp =  drv.FreeSpace / 1024 ' Get free space in kB.
' If the C: drive has less than 50000 kB of free space, create an alarm.
If (freesp < 50000) Then
  RAlm.CreateAlarm 0, "Insufficient free space on the C: drive.", "", Now, True, True,
    False, True, True, 0, "", , , ""
End If
Set fso = Nothing              ' Free the object referenced by fso.
Set drv = Nothing              ' Free the object referenced by drv.
```

3.6.4 RAlm.CurrentAlarms Procedure

RAlm.CurrentAlarms displays a window containing a list of current (i.e. active and/or unacknowledged) alarms.

Syntax

RAlm.CurrentAlarms

 **RAlm** Object

Example

```
If RTag.GetTagValue("System", "ShowAlarms") Then
  ' Display the list of current alarms depending on the value
  ' of the tag ShowAlarms from the device System.
  RAlm.CurrentAlarms
End If
```

3.6.5 RAlm.CurrentAlarmsByDevice Procedure

RAlm.CurrentAlarmsByDevice displays a window containing a list of current (i.e. active and/or unacknowledged) alarms belonging to a specified device.

Syntax

RAlm.CurrentAlarmsByDevice *Device*: **Variant**

Argument	Description
Device	The name or ID of the device.

RAlm Object

Example

```
' Display the list of current alarms belonging to the device PLC1.
RAlm.CurrentAlarmsByDevice "PLC1"
```

3.6.6 RAlm.DbAlarms Procedure

RAlm.DbAlarms displays a window containing a list of alarms stored in the alarm database (historical alarms).

Syntax

RAlm.DbAlarms *Unused*: **Integer**

Argument	Description
Unused	The value is no longer used.

Remarks

If the window contained a list of alarms restricted by a filter before calling the method, the filter is canceled.

RAlm Object

Example

```
' Display the list of historical alarms.
RAlm.DbAlarms 0
```

3.6.7 RAlm.DbAlarmsByDevice Procedure

RAlm.DbAlarmsByDevice displays a window containing a list of alarms stored in the alarm database (historical alarms), restricted to alarms belonging to a specified device.

Syntax

RAlm.DbAlarmsByDevice *Unused*: **Integer;** *Device*: **Variant**

Argument	Description
Unused	The value is no longer used.
Device	The name or ID of the device.

Remarks

If the window contained a list of alarms restricted by a filter before calling the method, the filter is canceled.

RAlm Object

Example

```
' Display the list of historical alarms belonging to the device PLC1.
RAlm.DbAlarmsByDevice 0, "PLC1"
```

3.6.8 RAlm.DbAlarmsByFilter Procedure

RAlm.DbAlarmsByFilter displays a window containing a list of alarms stored in the alarm database (historical alarms), restricted by a specified filter.

Syntax

RAlm.DbAlarmsByFilter *Unused*: **Integer;** *FilterName*: **String**

Argument	Description
Unused	The value is no longer used.
FilterName	The name of the filter.

Remarks

If the window contained a list of alarms restricted by a filter before calling the method, the filter is canceled and the specified filter is applied. To define filters for the alarm database, use the *Filter Editor* in the runtime software. The filters get stored to the profile (user profile in a visualization project, not in the operating system) of the user currently logged on to the runtime software. If the filter specified by *FilterName* is not found in the profile of the currently logged on user, the search is performed in the *Default* profile. If the filter is not found, the window will not be displayed.

RAlm Object

Example

```
' Display the list of historical alarms restricted by the filter PLC1_or_PLC2
' (displays only alarms belonging to the device PLC1 or PLC2).
RAlm.DbAlarmsByFilter 0, "PLC1_or_PLC2"
```

3.6.9 RAlm.DisableDeviceAlarms Procedure

RAlm.DisableDeviceAlarms disables all alarms, of a specified type, belonging to a specified device.

Syntax

RAlm.DisableDeviceAlarms *AlarmType*: **Byte;** *Device*: **Variant**

Argument	Description
AlarmType	Alarm type (Alarm Type Constants).
Device	The name or ID of the device.

Remarks

Alarms, which have been disabled, can be enabled again by calling the RAlm.EnableDeviceAlarms method.

RAlm Object

Example

```
' Disable all alarms belonging to the device PLC1.
RAlm.DisableDeviceAlarms -1, "PLC1"
```

3.6.10 RAlm.EnableDeviceAlarms Procedure

RAlm.EnableDeviceAlarms enables all alarms, of a specified type, belonging to a specified device.

Syntax

RAlm.EnableDeviceAlarms *AlarmType*: **Byte;** *Device*: **Variant**

Argument	Description
AlarmType	Alarm type (Alarm Type Constants).
Device	The name or ID of the device.

Remarks

Alarms, which have been enabled, can be disabled again by calling the RAlm.DisableDeviceAlarms method.

RAlm Object

Example

```
' Enable all alarms belonging to the device PLC1.
RAlm.EnableDeviceAlarms -1, "PLC1"
```

3.6.11 Alarm Type Constants

Value	Meaning
0	Alert.
1	Command.
2	System message.
-1	Includes all the above types.

RAlm Object

3.6.12 Alarm Triggering Condition Constants

Conditions that can trigger an alarm.

Value	Meaning
10	A change in the value of a tag (any change).
11	A change in the value of a tag (increment).
12	A change in the value of a tag (decrement).
20	The leading edge of a digital-type tag.
21	The trailing edge of a digital-type tag.
30	Exceeding the upper critical limit.
31	Exceeding the upper warning limit.
32	Falling below the lower warning limit.
33	Falling below the lower critical limit.
40	The value of a tag within the range.

 **RAlm** Object

3.7 RDb Object

The **RDb** object implements methods for operations on databases defined in a visualization project. The methods enable you to access historical data stored in the databases. You can also work with individual database tables through a TTable-type object returned by the RDb.CreateTableObject method.

Methods:

- **RDb.AppendRecord** Procedure
- **RDb.CreateTableObject** Function
- **RDb.GetTagHistValue** Function
- **RDb.GetTagStatistics** Procedure

3.7.1 RDb.AppendRecord Procedure

RDb.AppendRecord appends a new record to a specified database.

Syntax

RDb.AppendRecord *Database*: **Variant**

Argument	Description
Database	The name or ID of the database.

Remarks

In order for the method to work, the database's *Sampling* property must have a value of *Script controlled*.

RDb Object

Example

```
Const c_ArrayLen = 5
  ' Number of array-type tag elements
Dim ArrayIndex
  ' Index of array-type tag elements
```

```
For ArrayIndex = 0 to c_ArrayLen - 1
  RTag.SetTagValue "System", "TimeStamp", RTag.GetTagElementValue("PLC1",
"TimeStamp_Arr", ArrayIndex)
  RTag.SetTagValue "System", "Pressure1", RTag.GetTagElementValue("PLC1",
"Pressure1_Arr", ArrayIndex)
  RTag.SetTagValue "System", "Pressure2", RTag.GetTagElementValue("PLC1",
"Pressure2_Arr", ArrayIndex)
  RDb.AppendRecord "Pressures"
Next
```

3.7.2 RDb.CreateTableObject Function

RDb.CreateTableObject creates a new TTable-type object, which can be used for working with databases tables, and returns a reference to the object.

Syntax

```
RDb.CreateTableObject: TTable
```

Return values

The method returns a reference to the newly created TTable-type object.

Remarks

When using a TTable-type object, follow the rules for working with properties and methods of objects. When assigning the return value of **RDb.CreateTableObject** to a variable, the Set statement must be used. When you no longer need the TTable-type object, you should free it by assigning the constant `Nothing` to the variable using the Set statement.

 **RDb** Object

Example

```
Dim Table
Set Table = RDb.CreateTableObject ' Create a TTable-type object.
' Name of the database, as defined in a visualization project.
Table.DatabaseName = "Water"
If Table.OpenTable Then           ' If the current table can be opened.
  Table.Append                    ' Append a new record.
  ' Save current system date to the new record.
  Table.DateFieldValue = Date
  ' Save current system time to the new record.
  Table.TimeFieldValue = Time
```

```
  ' Save the value of the tag WaterTemperature from the device PLC1 to the new record.
  Table.SetFieldValue "PLC1", "WaterTemperature", RTag.GetTagValue("PLC1",
"WaterTemperature")
  Table.Post                    ' Write the new record to the table.
  Table.CloseTable              ' Close the table.
End If
Set Table = Nothing             ' Free the TTable-type object.
```

3.7.3 RDb.GetTagHistValue Function

RDb.GetTagHistValue returns a tag's historical value with time stamp (date and time) nearest to a specified date and time.

Syntax

RDb.GetTagHistValue(*DevName*, *TagName*, *DbName*: **String;** *ValTime*, *Tolerance*: **DateTime; ByRef** *RecTime*: **DateTime; ByRef** *Error*: **Variant): Variant**

Argument	Description
DevName	The name of the device that the tag belongs to.
TagName	The name of the tag.
DbName	The name of the database.
ValTime	Required time stamp of the value.
Tolerance	Time stamp tolerance.
RecTime	Time stamp of the database record found.
Error	Error code (The List of Error Codes Returned by Methods and Properties of Reliance-defined Objects).

Return values

If the call succeeds, the method returns the tag's historical value with time stamp nearest to the specified date and time.

If the call fails, the method returns **Empty**.

Remarks

The method searches in both current and archive files. The method attempts to find a record with time stamp nearest to the specified date and time in the range from (*ValTime* - *Tolerance*) to (*ValTime* + *Tolerance*). After calling the method, it is recommended to check the value of the *Error* argument to find out about the result of the operation.

RDb Object

Example

```
Dim Value, ValTime, Tolerance, RecTime, Error, Text
ValTime = Date + TimeSerial(6, 0, 0) ' Today, 6 a.m.
Tolerance = TimeSerial(0, 5, 0)      ' Time stamp tolerance +/- 5 minutes.
' Get the historical value of the tag WaterTemperature
' from the device PLC1 stored in the database Water.
Value = RDb.GetTagHistValue("PLC1", "WaterTemperature", "Water", ValTime, Tolerance,
RecTime, Error)
If Error = 0 Then
  Text = "The value at: " + CStr(RecTime) + " was: " + CStr(Value) + "."
Else
  Text = "Error (code " + CStr(Error) + ")."
End If
' Store information on the result of the operation
' to the tag DisplayResult from the device System.
RTag.SetTagValue "System", "DisplayResult", Text
```

3.7.4 RDb.GetTagStatistics Procedure

RDb.GetTagStatistics retrieves statistical information about a tag in a specified time range from a database.

Syntax

RDb.GetTagStatistics(*DevName*, *TagName*, *DbName*: **String;** *From*, *Till*: **DateTime;** **ByRef** *Min*, *Max*, *Sum*, *Ave*, *Count*, *Error*: **Variant)**

Argument	Description
DevName	The name of the device that the tag belongs to.
TagName	The name of the tag.
DbName	The name of the database.

From	The lower bound of the time range.
Till	The upper bound of the time range.
Min	The minimum value of the tag in the time range.
Max	The maximum value of the tag in the time range.
Sum	The sum of the values of the tag in the time range.
Ave	The average value of the tag in the time range.
Count	The number of database records in the time range.
Error	Error code (The List of Error Codes Returned by Methods and Properties of Reliance-defined Objects).

Remarks

The method searches in both current and archive databases. The method attempts to find and process the records with time stamp in the range specified by the *From* and *Till* arguments. After calling the method, it is recommended to check the value of the *Error* argument to find out about the result of the operation.

 **RDb** Object

Example

```
Dim Value, From, Till, AMin, AMax, ASum, AAve, ACount, Error, Text
From = Date + TimeSerial(6, 0, 0) ' From: today, 6 a.m.
Till = Date + TimeSerial(7, 0, 0) ' Till: today, 7 a.m.
' Retrieve statistical information about the tag WaterTemperature
' from the device PLC1 stored in the database Water.
Value = RDb.GetTagStatistics("PLC1", "WaterTemperature", "Water", From, Till, AMin,
AMax, ASum, AAve, ACount, Error)
If Error = 0 Then
  Text = "Minimum: " + CStr(AMin) + " Maximum: " + CStr(AMax) + " Sum: " + CStr(ASum)
+ " Average: " + CStr(AAve) + " Record count: " + CStr(ACount) + "."
Else
  Text = "Error (code " + CStr(Error) + ")."
End If
```

```
' Store information on the result of the operation
' to the tag DisplayResult from the device System.
RTag.SetTagValue "System", "DisplayResult", Text
```

3.8 RDev Object

The **RDev** object implements methods for working with devices.

Methods:

- **RDev.ConnectToCommDriver** Procedure
- **RDev.SendCustomData** Procedure
- **RDev.ReceiveCustomDataReply** Procedure

3.8.1 RDev.ConnectToCommDriver Procedure

RDev.ConnectToCommDriver connects the runtime software to a specified communication driver.

Syntax

RDev.ConnectToCommDriver *ProgID*: **String;** *Computer*: **Variant**

Argument	Description
ProgID	A unique identifier of the communication driver.
Computer	The name or ID (as defined in a visualization project) of the computer hosting the communication driver. A special value of "" means the local computer (i.e. the computer on which the runtime software is running).

Remarks

This method is sometimes used to reconnect the runtime software to a communication driver (e.g. if the driver was terminated and could not be restarted automatically by the runtime software).

- **RDev** Object

Example

```
' Connect the runtime software to Teco OPC Server on the local computer.
RDev.ConnectToCommDriver "TECO.DA2", ""
```

3.8.2 RDev.SendCustomData Procedure

Sends custom data to a device(s) via a specified communication driver.

Syntax

RDev.SendCustomData *ProgID*: **String,** *Computer*: **Variant,** *Params:* **Variant,** *Data:* **Variant**

Argument	Description
ProgID	A unique identifier (as registered in the Windows registry) of the communication driver's COM interface.
Computer	The name or ID (as defined in a visualization project) of the computer hosting the communication driver. A special value of "" means the local computer (i.e. the computer on which the runtime software is running).
Params	Communication parameters.
Data	Data to be send.

Remarks

This method is sometimes used to send data from scripts.

RDev Object

Example

```
dim Params
' The first parameter represents a target where the communication packet should be
sent to. 1 - to a communication channel, 2 - to a device. (only code 2 - device - is
supported)
```

```
' The second parameter represents the channel's or device's ID depending on the value
of the 1st parameter. In this example, it is the device's ID.
Params = array( 2,2)
' Send the user-defined communication packet to the Sauter device.
RDev. SendCustomData "R_DrvSauter. dll","",Params,"P00101N/"
```

3.8.3 RDev.RDev.ReceiveCustomDataReply Procedure

Receives a reply (if any) to custom data sent to a device(s) via a specified communication driver.

Syntax

RDev.ReceiveCustomDataReply *ProgID*: **String,** *Computer*: **Variant,** *Params:* **Variant, ByRef** *DataReply:* **Variant**

Argument	Description
ProgID	A unique identifier (as registered in the Windows registry) of the communication driver's COM interface.
Computer	The name or ID (as defined in a visualization project) of the computer hosting the communication driver. A special value of "" means the local computer (i.e. the computer on which the runtime software is running).
Params	Communication parameters.
DataReply	Reply to a data, parameter passed by reference.

Remarks

This method is sometimes used to receive reply to a data.

RDev Object

Example

```
dim Params, Data
' The first parameter represents a source from which the communication packet should be received. 1 - from a communication channel, 2 - from a device. (only code 2 - device - is supported)
' The second parameter represents the channel's or device's ID depending on the value of the 1st parameter. In this example, it is the device's ID.
Params = array(2,2)
' Receive a reply to the user-defined communication packet sent to the Sauter device.
RDev.ReceiveCustomDataReply "R_DrvSauter.dll","",Params,Data
```

3.9 RError Object

The **RError** object provides information on the result of the most recent call to a method or access to a property of any Reliance-defined object with the exception of the **RError** object itself.

Properties:

- **RError.Code** Property
- **RError.Description** Property

Other:

- The List of Error Codes Returned by Methods and Properties of *Reliance-defined Objects*

3.9.1 RError.Code Property

RError.Code returns the error code of the most recent call to a method or access to a property of any Reliance-defined object with the exception of the **RError** object itself.

Syntax

```
RError.Code: Integer
```

Remarks

The property is read-only.

Value	Meaning
0	The most recent method call or property access was successful.
>0	The code of the error (The List of Error Codes Returned by Methods and Properties of Reliance-defined Objects).

- **RError** Object

Example

```
If RError.Code > 0 Then
  ' Store the error description
  ' to the tag DisplayResult from the device System.
  RTag.SetTagValue "System", "DisplayResult", RError.Description
End If
```

3.9.2 RError.Description Property

RError.Description returns the error description of the most recent call to a method or access to a property of any Reliance-defined object with the exception of the **RError** object itself.

Syntax

```
RError.Description: String
```

Remarks

The property is read-only. The List of Error Codes Returned by Methods and Properties of Reliance-defined Objects.

RError Object

Example

```
If RError.Code > 0 Then
  ' Store the error description
  ' to the tag DisplayResult from the device System.
  RTag.SetTagValue "System", "DisplayResult", RError.Description
End If
```

3.9.3 The List of Reliance-defined Objects Error Codes

Error codes common to all objects

Code (RError.Code)	**Description** (RError.Description)	**Explanation**
0	Success.	The call to a method or access to a property was successful (i.e. no error occurred).

1	Unknown error.	An unknown error occurred (i.e. an error in a block of code where an error is expected but there is no way to find out exactly what error occurred).
2	Unexpected error.	An unexpected error occurred (i.e. an error in a block of code where an error is not expected, e.g. when calling the **RDb. GetTagHistValue** method).
3	Service is not supported.	The service provided by the method called is not supported.
100	Device not found.	The device was not found. The name of the device was not correctly specified or the device is not accessible to the computer on which the script was executed.
101	Tag not found.	The tag was not found. The name of the tag was not correctly specified.
102	Database not found.	The database was not found. The name of the database was not correctly specified or the database is not accessible to the computer on which the script was executed.
500	Cannot create directory.	The directory cannot be created.

RError Object

Error codes specific to TTable-type objects

Code (RError. Code)	**Description** (RError. Description)	**Explanation**

10001	Tag not included in database definition.	The database defined in a visualization project does not contain a field linked to the tag (i.e. the tag is not configured to be logged to the database).
10002	Missing database name.	The *DatabaseName* property is blank (e.g. because it has not been assigned a value).
10003	Missing archive name.	The *ArchiveName* property is blank (e.g. because it has not been assigned a value).
10010	Table does not exist.	The database table does not exist.
10011	Table already exists.	The database table already exists.
10012	Table not active.	The database table is not open.
10013	Table not in edit or insert mode.	The database table is not in edit or insert mode.
10014	Table is busy.	The database table is currently being accessed by another program.
10015	Cannot open table.	The database table cannot be opened.
10016	TTable object does not support working with SQL databases.	The TTable object does not allow for working with tables in SQL databases.
10030	Field not found in table.	The database field was not found in the database table although the tag is configured to be logged to the database.
10031	Invalid field value.	The returned value of the database field is invalid (when reading the value of the field).

10032	Invalid value for field.	The value to be assigned to the database field is invalid (when writing the value of the field).

TTable-type Objects

Error codes specific to the RTag object

Code (RError.Code)	**Description** (RError.Description)	**Explanation**
11001	Invalid tag value.	The returned value of the tag is invalid (e.g. because the value has not yet been returned by the communication driver).
11002	Incorrect tag type.	The tag is of an incorrect data type (e.g. the value of an array-type tag cannot be returned by calling the **RTag.GetTagValue** method).
11003	Cannot send commands.	The tag cannot be written (e.g. because a visualization project runs in a view-only version of the runtime software).
11004	Cannot assign value to tag.	The tag cannot be assigned the specified value because the data type of the value is not assignment compatible with the data type of the tag.
11005	Array-type tag required.	The method called requires an array-type tag.
11006	Non-array-type tag required.	The method called requires a non-array-type tag.
11007	Incompatible tag types.	The data types of the tags are not compatible (e.g. when calling the **RTag.MoveTagValue** method).
11008	Unsupported tag type.	The method called does not support the data type of the tag.

11009	Cannot determine tag value.	The value of the tag cannot be determined.
11010	Array index out of bounds.	The specified index into the array-type tag is out of bounds (e.g. when calling the **RTag.MoveTagElementValues** method).
11011	Element count out of bounds.	The specified number of elements exceeds the size of the array-type tag (e.g. when calling the **RTag.MoveTagElementValues** method).
11012	Zero elements to move.	The specified number of elements to move is zero (when calling the **RTag.MoveTagElementValues** method).
11013	Device driver not available.	The device's communication driver is not available (e.g. is not running).
11014	Device currently not connected.	A connection to the device is currently not established (e.g. because a connection to the server computer has not been established).

RTag Object

Error codes specific to the RSys object

Code (RError.Code)	**Description** (RError.Description)	**Explanation**
12001	Window not found.	The project window was not found. The name of the window was not correctly specified or the window is not accessible to the computer on which the script was executed (e.g. when calling the **RSys.ActivateWindow** method).

12002	Program file name cannot be empty.	The file name of the program to be run cannot be blank (e.g. when calling the **RSys.ExecApp** method).
12003	Sound file name cannot be empty.	The file name of the sound to be played cannot be blank (e.g. when calling the **RSys.PlaySound** method).
12010	File does not exist.	The file does not exist.
12011	Source file does not exist.	The source file does not exist (e.g. when calling the **RSys.CopyFile** method).
12012	Destination file already exists.	The destination file already exists.
12013	Cannot copy a file to itself.	The names of the source and destination files are identical. The operation cannot be performed.

RSys Object

Error codes specific to the RDb object

Code (RError.Code)	**Description** (RError.Description)	**Explanation**
13001	Database does not exist.	None of the tables of the database exist (neither current nor archive).
13002	Tag not stored in database.	The tag was not found in any of the tables belonging to the database (the tag is not configured to be logged to the database).
13003	Incorrect tag type.	The tag is of an incorrect data type (e.g. a string-type tag when calling the **R.GetTagStatistics** method).
13004	Cannot open database.	None of the tables of the database could be opened.

13005	No data available.	There are no records in the database in the specified time range.

RDb Object

Error codes specific to the RScr object

Code (RError.Code)	**Description** (RError.Description)	**Explanation**
14001	Script not found.	The script was not found. The name of the script was not correctly specified.

RScr Object

Error codes specific to the RUser object

Code (RError.Code)	**Description** (RError.Description)	**Explanation**
15001	User not found.	The user was not found. The name of the user was not correctly specified.
15002	No user logged on.	No user is currently logged on to the runtime software.
15003	Invalid access rights.	The specified set of access rights is invalid, since one or more rights have invalid names.

RUser Object

Error codes specific to the RAlm object

Code (RError.Code)	**Description** (RError.Description)	**Explanation**

16001	Invalid alarm type.	The specified alarm type is invalid, since it is not within the range 0 to 2.

RAlm Object

Error codes specific to the RInet object

Code (RError.Code)	**Description** (RError.Description)	**Explanation**
17001	Error initializing SMTP.	An error occurred while initializing the SMTP routines (routines for sending E-mail messages).
17002	Incomplete SMTP configuration.	The SMTP configuration is incomplete (some properties have not been specified).
17003	Error connecting to SMTP server.	An error occurred while connecting to the SMTP (E-mail) server.
17004	Error sending E-mail.	An error occurred while sending the E-mail message.

RInet Object

Error codes specific to the RModem object

Code (RError.Code)	**Description** (RError.Description)	**Explanation**
18001	SMS driver not initialized.	The GSM SMS driver has not been initialized.
18002	Error sending SMS.	An error occurred while sending the SMS message.
18003	Error dialing phone number.	An error occurred while dialing the phone number.

18004	SMS not found.	The SMS message was not found by the GSM SMS driver.
18005	Error sending AT command.	An error occurred while sending the AT command.

RModem Object

3.10 RInet Object

The **RInet** object implements methods for sending E-mail messages. In order for the methods to be operational, you have to correctly configure computer properties related to sending E-mail messages (configure the *SMTP server*, *Port number*, *Connection timeout* and *Sender address* properties on the *E-mail* page in the **Project Structure Manager**).

Methods:

RInet.SendMail Function

3.10.1 RInet.SendMail Function

RInet.SendMail sends an E-mail message.

Syntax

RInet.SendMail(*DestAddress***,** *Subject***,** *Text***,** *FileName***: String): Boolean**

Argument	Description
DestAddress	The list of recipients (E-mail addresses) separated by a semicolon.
Subject	The subject of the message.
Text	The text of the message.
FileName	The list of attachments (file names) separated by a semicolon.

Return values

Value	Meaning
True	The message was sent.
False	An error occurred while sending the message (information on the error is logged to the alarm database).

Remarks

If you want to send an E-mail message with no attachments, pass an empty string ("") as the last argument.

 **RInet** Object

Example

```
' Send two files to the address reliance@hotmail.com.
If RInet.SendMail("reliance@hotmail.com", "Data files", "Data files Data1.txt and
Data2.txt.", "C:\Data\Data1.txt;C:\Data\Data2.txt") Then
  ' Store information on the result of the operation
  ' to the tag DisplayResult from the device System.
  RTag.SetTagValue "System", "DisplayResult", "The message was sent."
End If
```

3.11 RModem Object

The **RModem** object implements methods for sending and receiving SMS messages through a GSM modem connected to the computer. In order for the methods to be operational, you have to correctly configure computer properties related to GSM modems (activate the *Start SMS driver* property and configure the *GSM device type*, *Communication options* and *SMS service center number* properties on the *SMS* page in the **Project Structure Manager**).

Methods:

- **RModem.GSMGetSMSStatus** Function
- **RModem.GSMSendSMS** Function
- **RModem.GSMSendSMSEx** Function

Other:

- The List of Error Codes (CMS) According to the GSM 07.05 Standard

3.11.1 RModem.GSMSendATCommand Function

RModem.GSMSendATCommand sends a specified AT command to the GSM SMS driver.

Syntax

RModem.GSMSendATCommand(*Command***: String): Boolean**

Argument	Description
Command	AT command

Return values

Value	Meaning
True	The AT command was placed in a queue by the GSM SMS driver.
False	An error occurred.

Remarks

The method passes the AT command to the GSM SMS driver which places it in an internal queue. The AT command is passed to the GSM modem when all previous requests in the queue are processed.

RModem Object

Example

```
Dim SCNumber, ATCommand
' The service centre number is stored in the tag SCNumber from the device System.
SCNumber = RTag.GetTagValue("System", "SCNumber")
ATCommand = "AT+CSCA=""" & SCNumber & """"
If RModem.GSMSendATCommand(ATCommand) Then
  ' ...
End If
```

3.11.2 RModem.GSMGetSMSStatus Function

RModem.GSMGetSMSStatus returns the status of a SMS message previously sent by calling the RModem.GSMSendSMSEx method.

Syntax

RModem.GSMGetSMSStatus(*ID*: **Integer; ByRef** *Text*, *PhoneNumber*, *Status*, *ErrorCode*: **Variant): Boolean**

Argument	Description
ID	The unique identifier of the message returned by the RModem.GSMSendSMSEx method.
Text	The text of the message.
PhoneNumber	The phone number of the message's recipient.
Status	The status of the message (see the following table).
ErrorCode	The code of an error which might occur while sending the message (The List of Error Codes (CMS) According to the GSM 07.05 Standard).

Value	Meaning
0	The message is waiting in the queue.
1	The message was successfully sent.
2	An error occurred while sending the message. The code of the error is returned in the *ErrorCode* argument.
3	The status of the message is not available.

Return values

Value	Meaning
True	The status of the message is available.
False	The status of the message is not available.

 **RModem** Object

Example

```
Dim PhoneNumber, Text, Status, ErrCode, ID
' The identifier of the most recently sent message is stored
' in the tag LastSMSID from the device System.
ID = RTag.GetTagValue("System", "LastSMSID")
' Retrieve the status of the message.
RModem.GSMGetSMSStatus ID, Text, PhoneNumber, Status, ErrCode
' Store the status of the message to the
' tag LastSMSStatus from the device System.
RTag.SetTagValue "System", "LastSMSStatus", Status
' Store the error code of the message to the
' tag LastSMSErrCode from the device System.
RTag.SetTagValue "System", "LastSMSErrCode", ErrCode
```

3.11.3 RModem.GSMSendSMS Function

RModem.GSMSendSMS sends a SMS message to a specified phone number by passing it to the GSM SMS driver.

Syntax

RModem.GSMSendSMS(*PhoneNumber***, *Text*: String): Boolean**

Argument	Description
PhoneNumber	The phone number of the message's recipient.
Text	The text of the message.

Return values

Value	Meaning
True	The message was encoded and placed in a queue by the GSM SMS driver.
False	An error occurred.

Remarks

The method passes the message to the GSM SMS driver which places it in an internal queue. However, there is no way to find out whether the message was actually sent. If you want to retrieve the result of sending the message, use the RModem.GSMSendSMSEx method instead.

RModem Object

Example

```
Dim PhoneNumber, Text
' The phone number is stored in the tag Number from the device System.
PhoneNumber = RTag.GetTagValue("System", "Number")
' The text of the message is stored in the tag Text from the device System.
Text = RTag.GetTagValue("System", "Text")
```

```
If RModem.GSMSendSMS(PhoneNumber, Text) Then
  ' ...
End If
```

3.11.4 RModem.GSMSendSMSEx Function

RModem.GSMSendSMSEx sends a SMS message to a specified phone number by passing it to the GSM SMS driver and returns a unique identifier for the message.

Syntax

RModem.GSMSendSMSEx(*PhoneNumber*, *Text*: **String; ByRef** *ID*: **Integer): `Boolean`**

Argument	Description
PhoneNumber	The phone number of the message's recipient.
Text	The text of the message.
ID	The identifier of the message.

Return values

Value	Meaning
True	The message was encoded and placed in a queue by the GSM SMS driver.
False	An error occurred.

Remarks

The method passes the message to the GSM SMS driver which places it in an internal queue and returns a unique identifier (a number unique within an instance of the driver) for the message. The identifier can later be used to retrieve the status of the message by passing it to theRModem.GSMGetSMSStatus method. Information on the status of sending a message is stored in the driver for a time period determined by the driver's settings (the default time period is 24 hours).

RModem Object

Example

```
Dim PhoneNumber, Text, ID
' The phone number is stored in the tag Number from the device System.
PhoneNumber = RTag.GetTagValue("System", "Number")
' The text of the message is stored in the tag Text from the device System.
Text = RTag.GetTagValue("System", "Text")
If RModem.GSMSendSMSEx(PhoneNumber, Text, ID) Then
  ' Store the value of ID so that it can later be passed
  ' to the RModem.GSMGetSMSStatus method.
  RTag.SetTagValue "System", "LastSMSID", ID
End If
```

3.11.5 The List of Error Codes (CMS) According to GSM 07.05 Standard

Code	Meaning
1	Unassigned (unallocated) number.
8	Operator determined barring.
10	Call barred.
21	Short message transfer rejected.
27	Destination out of service.
28	Unidentified subscriber.
29	Facility rejected.
30	Unknown subscriber.
38	Network out of order.
41	Temporary failure.
42	Congestion.
47	Resources unavailable, unspecified.
50	Requested facility not subscribed.
69	Requested facility not implemented.

81	Invalid short message transfer reference value.
95	Invalid message, unspecified.
96	Invalid mandatory information.
97	Message type non-existent or not implemented.
98	Message not compatible with short message protocol state.
99	Information element non-existent or not implemented.
111	Protocol error, unspecified.
127	Interworking, unspecified.
128	Telematic interworking not supported.
129	Short message Type 0 not supported.
130	Cannot replace short message.
143	Unspecified TP-PID error.
144	Data coding scheme (alphabet) not supported.
145	Message class not supported.
159	Unspecified TP-DCS error.
160	Command cannot be actioned.
161	Command unsupported.
175	Unspecified TP-Command error.
176	TPDU not supported.
192	SC busy.
193	No SC subscription.
194	SC system failure.
195	Invalid SME address.

196	Destination SME barred.
197	SM Rejected-Duplicate SM.
198	TP-VPF not supported.
199	TP-VP not supported.
208	D0 SIM SMS storage full.
209	No SMS storage capability in SIM.
210	Error in MS.
211	Memory Capacity Exceeded.
212	SIM Application Toolkit Busy.
213	SIM data download error.
255	Unspecified error cause.
300	ME failure.
301	SMS service of ME reserved.
302	Operation not allowed.
303	Operation not supported.
304	Invalid PDU mode parameter.
305	Invalid text mode parameter.
310	SIM not inserted.
311	SIM PIN required.
312	PH-SIM PIN required.
313	SIM failure.
314	SIM busy.
315	SIM wrong.
316	SIM PUK required.
317	SIM PIN2 required.

318	SIM PUK2 required.
320	Memory failure.
321	Invalid memory index.
322	Memory full.
330	SMSC address unknown.
331	No network service.
332	Network timeout.
340	NO +CNMA ACK EXPECTED.
500	Unknown error.
512	User abort.
513	Unable to store.
514	Invalid status.
515	Invalid character in address string.
516	Invalid length.
517	Invalid character in pdu.
518	Invalid parameter.
519	Invalid length or character.
520	Invalid character in text.
521	Timer expired.

RModem Object

3.12 RScr Object

The **RScr** object implements methods for operations on scripts defined in a visualization project.

The methods enable you to enable/disable scripts, retrieve parameters passed to a script and retrieve information on scripts.

Methods:

- **RScr.DisableScript** Procedure
- **RScr.EnableScript** Procedure
- **RScr.ExecScript** Procedure
- **RScr.GetCurrentScriptData** Function
- **RScr.GetCurrentScriptDataEx** Function
- **RScr.GetScriptInfo** Function
- **RScr.GetScriptText** Function

Events:

- Basic Events
- Events Triggered by a Component
- Events Triggered by an Alarm
- Events Triggered by a SMS Message

3.12.1 RScr.DisableScript Procedure

RScr.DisableScript disables a script.

Syntax

RScr.DisableScript *Script*: **Variant**

Argument	Description
Script	The name or ID of the script.

Remarks

Calling this method disables the script only for the time a visualization project is running. After terminating the project, the information is lost. When the project is started next time, the script is again in its initial state, i.e. either enabled or disabled, depending on the value of the *Enable execution* property (*Reliance Design* > *Managers* > *Script Manager* > script properties > the *Basic* page).

RScr Object

Example

```
If RSys.GetComputerName = "PC1" Then
  ' Disable the script Script1 if the current script
  ' is executing on the computer PC1.
  RScr.DisableScript "Script1"
Else
  ' Enable the script Script1.
  RScr.EnableScript "Script1"
End If
```

3.12.2 RScr.EnableScript Procedure

RScr.EnableScript enables a script.

Syntax

RScr.EnableScript *Script*: **Variant**

Argument	Description
Script	The name or ID of the script.

Remarks

Calling this method disables the script only for the time a visualization project is running. After terminating the project, the information is lost. When the project is started next time, the script is again in its initial state, i.e. either enabled or disabled, depending on the value of the *Enable execution* property (*Reliance Design* > *Managers* > *Script Manager* > script properties > the *Basic* page).

RScr Object

Example

```
If RSys.GetComputerName = "PC1" Then
  ' Disable the script Script1 if the current script
  ' is executing on the computer PC1.
  RScr.DisableScript "Script1"
Else
  ' Enable the script Script1.
  RScr.EnableScript "Script1"
End If
```

3.12.3 RScr.ExecScript Procedure

RScr.ExecScript places a script in a queue of scripts to be executed.

Syntax

RScr.ExecScript *Script*: **Variant;** *PriorExec*: **Boolean**

Argument	Description
Script	The name or ID of the script.
PriorExec	Determines whether to execute the script prior to other scripts (see the following table).

Value	Meaning
True	The script is placed in the prior script queue to a position that depends on the script's *priority* (*Reliance Design* > *Managers* > *Script Manager* > script properties > the *Advanced* page). Placing the script in the prior script queue ensures that the script is executed prior to all scripts placed in the standard script queue.
False	The script is placed in the standard script queue to a position that depends on the script's *priority*.

Remarks

The script is placed in the appropriate script queue and can be executed only after the current script finishes execution. To execute a script's code immediately, use the Execute statement in combination with the RScr.GetScriptText method.

RScr Object

Example

```
' Place the script Script1 in the prior script queue.
RScr.ExecScript "Script1", True
```

3.12.4 RScr.GetCurrentScriptData Function

RScr.GetCurrentScriptData returns data passed as a parameter to the current script.

Syntax

RScr.GetCurrentScriptData(ByRef *Data***: Variant): Boolean**

Argument	Description
Data	The data passed to the current script.

Return values

Value	Meaning
True	The *Data* argument has been assigned the value of the data passed to the current script.
False	The *Data* argument has not been assigned a value, since no data has been passed to the current script.

Remarks

If the current script has been triggered by an alarm (i.e. start, end, or acknowledgment), the method retrieves the text of the alarm.

This method is obsolete and is provided only for backward compatibility. For new applications, use the RScr.GetCurrentScriptDataEx method.

RScr Object

Example

```
' This script is intended to be triggered when an alarm is generated.
Dim Data
' If the Data argument has been assigned a value.
If RScr.GetCurrentScriptData(Data) Then
  ' Send an E-mail containing the text of the alarm.
  If RInet.SendMail("reliance@hotmail.com", "Alarm generated", Data, "") Then
    ' ...
  End If
End If
```

3.12.5 RScr.GetCurrentScriptDataEx Function

RScr.GetCurrentScriptDataEx returns data passed as a parameter to the current script.

Syntax

RScr.GetCurrentScriptDataEx(ByRef *Data***: Variant): Boolean**

Argument	Description
Data	The data passed to the current script.

Return values

Value	Meaning
True	The *Data* argument has been assigned the value of the data passed to the current script.
False	The *Data* argument has not been assigned a value, since no data has been passed to the current script.

Remarks

If the method returns **True**, the *Data* argument references an object with the following properties:

Property	Type
SenderName	**String**
StrPar1	**String**
StrPar2	**String**
StrPar3	**String**
StrPar4	**String**
StrPar5	**String**
StrPar6	**String**
StrPar7	**String**
StrPar8	**String**
DoublePar1	**Double**
DoublePar2	**Double**
DatePar1	**DateTime**
DatePar2	**DateTime**
DatePar3	**DateTime**
DatePar4	**DateTime**
IntPar1	**Integer**
IntPar2	**Integer**
IntPar3	**Integer**
IntPar4	**Integer**
BytePar1	**Byte**
BytePar2	**Byte**

BytePar3	**Byte**
BytePar4	**Byte**
BoolPar1	**Boolean**
BoolPar2	**Boolean**
BoolPar3	**Boolean**
BoolPar4	**Boolean**
BoolPar5	**Boolean**

The meaning of individual properties depends on the event that triggered the script:

Basic Events

Events Triggered by a Component

Events Triggered by an Alarm

Events Triggered by a SMS Message

Events Triggered by a Thin Client Request

RScr Object

Example 1

```
' This script is intended to be triggered by clicking a component.
Dim Data
' If the Data argument has been assigned a value.
If RScr.GetCurrentScriptDataEx(Data) Then
  ' If the user clicked a component that runs this script
  ' with the parameter equal to 3.
  If Data.IntPar1 = 3 Then
    ' ...
  End If
End If
```

Example 2

```
' This script is intended to be triggered when an alarm is generated.
Dim Data
' If the Data argument has been assigned a value.
```

```
If RScr.GetCurrentScriptDataEx(Data) Then
  ' Send an E-mail containing the text of the alarm.
  If RInet.SendMail("reliance@hotmail.com", "Alarm generated", Data.StrPar1, "") Then
    ' ...
  End If
End If
```

Example 3

```
' This script is intended to be triggered by receiving a SMS message.
Dim Data
' If the Data argument has been assigned a value
' we can to access individual properties.
If Rscr.GetCurrentScriptDataEx(Data) Then
  RTag.SetTagValue "System", "Rec_Data", Data.StrPar1    ' Data.
  RTag.SetTagValue "System", "Rec_Text", Data.StrPar2    ' Text.
  RTag.SetTagValue "System", "Rec_Number", Data.StrPar3 ' Number.
  ' The time assigned by the SMS service center.
  RTag.SetTagValue "System", "Rec_SCTime", Data.StrPar4
  ' The time of receiving the message by the GSM SMS driver.
  RTag.SetTagValue "System", "Rec_Time", CStr(Data.DatePar1)
End If
```

Příklad 4

```
' This script is intended to be triggered by thin client request.
Dim Data
' If the Data argument has been assigned a value
' we can to access individual properties.
If Rscr.GetCurrentScriptDataEx(Data) Then
  ' Request Type: Timeout = 0, Connect = 1, Disconnect = 2, User Logon = 3, User
Logout = 4
  Select Case Data.IntPar1
    Case 0
      RTag.SetTagValue "System", "Request_Type", "Disconnect (Expired Session)"
    Case 1
      RTag.SetTagValue "System", "Request_Type", "Connect"
    Case 2
      RTag.SetTagValue "System", "Request_Type", "Disconnect"
    Case 3
      RTag.SetTagValue "System", "Request_Type", "User Logon"
    Case 4
      RTag.SetTagValue "System", "Request_Type", "User Logout"
  End Select
  ' Thin Client Type: Reliance Web Client = 0, Reliance Mobile Client = 1
  Select Case Data.IntPar2
```

```
    Case 0
      RTag.SetTagValue "System", "Client_Type", "Reliance Web Client"
    Case 1
      RTag.SetTagValue "System", "Client_Type", "Reliance Mobile Client"
    Case 2
      RTag.SetTagValue "System", "Client_Type", "Reliance Smart Client"
  End Select
  RTag.SetTagValue "System", "Session_Name", Data.StrPar1     ' Unique Session
Identifier.
  RTag.SetTagValue "System", "IP", Data.StrPar2               ' Client IP Address.
  RTag.SetTagValue "System", "Software_Version", Data.StrPar3 ' Client Version.
  RTag.SetTagValue "System", "Computer_Name", Data.StrPar4    ' Computer Name
(Configuration).
  RTag.SetTagValue "System", "User_Name", Data.StrPar5        ' User Name.
  RTag.SetTagValue "System", "User_Agent", Data.StrPar6       ' Information on Web
browser (User-Agent header). Only for Reliance Smart Client.
End If
```

3.12.6 RScr.GetScriptInfo Function

RScr.GetScriptInfo returns information on a script.

Syntax

RScr.GetScriptInfo(*Script***: Variant; ByRef** *Enabled*, *LastExecStartTime*, *LastExecEndTime*, *LastForcedTerminTime*, *ForcedTerminCou*, *ExecErrorCount***: Variant): Boolean**

Argument	Description
Script	The name or ID of the script.
Enabled	Determines whether the script is enabled.
LastExecStartTime	The date and time that the script last started execution.
LastExecEndTime	The date and time that the script last finished execution.
LastForcedTerminTime	The date and time that the script was last forcibly terminated.
ForcedTerminCount	The number of forcible terminations of the script during the time a visualization project is running.

ExecErrorCount	The number of errors (other than syntax errors) while executing the script during the time a visualization project is running.

Return values

Value	Meaning
True	The script was found and the returned information is valid.
False	The script was not found and the returned information is not valid.

RScr Object

Example

```
Dim Enabled, LastExecStartTime, LastExecEndTime, LastForcedTerminTime,
ForcedTerminCount, ExecErrorCount
' If the script Script1 was found.
If RScr.GetScriptInfo("Script1", Enabled, LastExecStartTime, LastExecEndTime,
LastForcedTerminTime, ForcedTerminCount, ExecErrorCount) Then
  If Not Enabled Then           ' If the script is not enabled.
    RScr.EnableScript "Script1" ' Enable the script.
  End If
End If
```

3.12.7 RScr.GetScriptText Function

RScr.GetScriptText returns the text (program code) of a script.

Syntax

RScr.GetScriptText(*Script***: Variant): String**

Argument	Description
Script	The name or ID of the script.

Remarks

This method can be used in combination with the Execute statement to execute a script's code immediately.

RScr Object

Example

```
Dim ScriptCode
' Place the script Script2 in the prior script queue.
RScr.ExecScript "Script2", True
ScriptCode = RScr.GetScriptText("Script1")
' Retrieve the text (program code) of the script Script1
' and execute it using the Execute statement.
' Thus, Script1 executes sooner than Script2.
Execute ScriptCode
```

3.12.8 Basic Events

Basic events include the following events:

- clicking or double-clicking a window
- loading, activating, deactivating, closing, and freeing a window
- starting, interrupting, and restoring communication to a device (e.g. PLC)
- starting and terminating communication on a network connection between instances of the runtime software
- receiving data for a data table from a data server

When a script is triggered by a basic event, an integer parameter configured for the event is passed to the script. In addition, the complete name of the object (window, device, server connection, data table) that triggered the event is passed to the script.

The meaning of the properties of the object returned by the RScr.GetCurrentScriptDataEx method:

Property	Meaning
SenderName	The complete name of the object that triggered the event.

IntPar1	The integer parameter passed to the script.

3.12.9 Events Triggered by a Component

When a script is triggered by clicking or double-clicking a component, information on the component is passed to the script.

The meaning of the properties of the object returned by the RScr.GetCurrentScriptDataEx method:

Property	Meaning
SenderName	The complete name of the component, i.e. window name/component name (e.g. *Window1/Button1*)
IntPar1	User defined parameter (integer parameter as with basic events)
StrPar1	The name of the component.
StrPar2	The name of the window containing the component.
StrPar3	The complete name of the tag (i.e. device name/tag name) referenced by the container through which the component is inserted into a window as part of a window template.
StrPar4	The complete name of the component's main tag (i.e. device name/tag name).
StrPar5	Unique session identifier (only if the script is run from a thin client).
IntPar2	The ID of the window containing the component.

Remarks

The separator of the window and component name in the *SenderName* property depends on a setting in the *Project Options* dialog (*Project* > *Objects*). The same applies to the separator used in the *StrPar3* and *StrPar4* properties.

3.12.10 Events Triggered by an SMS Message

When a script is triggered by receiving a SMS message, information on the message is passed to the script.

The meaning of the properties of the object returned by the RScr.GetCurrentScriptDataEx method:

Property	Meaning
SenderName	The name of the logical computer (based on the project) on which the project is running.
StrPar1	Data in the PDU format (before decoding).
StrPar2	The text of the message.
StrPar3	The phone number of the sender.
StrPar4	The time (in text form) assigned by the SMS service center.
DatePar1	The time of receiving the message by the GSM SMS driver.

3.12.11 Events Triggered by an Alarm

When a script is triggered by an alarm (i.e. start, end, or acknowledgment), information on the alarm is passed to the script.

The meaning of the properties of the object returned by the RScr.GetCurrentScriptDataEx method:

Property	Meaning

SenderName	The complete name of the alarm, i.e. device name/alarm name (e.g. *Modbus1/MotorFault*)
StrPar1	Date and time (in local time) when this instance of the alarm was generated followed by the text of the alarm.
StrPar2	The name of the device to which the alarm belongs.
StrPar3	The name of the tag related to the alarm.
StrPar4	The name of the alarm.
StrPar5	The alias of the device to which the alarm belongs.
StrPar6	The compound alias of the tag related to the alarm.
StrPar7	The alias of the alarm.
StrPar8	The text of the alarm.
IntPar1	The ID of the alarm.
IntPar2	The ID of the device to which the alarm belongs.
IntPar3	The ID of the tag related to the alarm.
IntPar4	The access rights required for acknowledging the alarm.
BytePar1	Alarm type (Alarm Type Constants).
BytePar2	The occurrence that generated the alarm (Alarm Triggering Condition Constants).
BytePar3	The priority of the alarm.
BoolPar1	Determines whether to display the alarm in the list of current alarms.
BoolPar2	Determines whether to log the alarm to the alarm database.
BoolPar3	Determines whether it is required that the alarm be acknowledged by the user (operator).
BoolPar4	Determines whether this instance of the alarm is active.
BoolPar5	Determines whether this instance of the alarm has been acknowledged.

DatePar1	Date and time (in UTC) when this instance of the alarm was generated.
DatePar2	Date and time (in UTC) when the most recent instance of the alarm was generated.
DatePar3	Date and time (in UTC) when this instance of the alarm ended.
DatePar4	Date and time (in UTC) when this instance of the alarm was acknowledged.

Remarks

The separator of the device and tag name in the *SenderName* property depends on a setting in the *Project Options* dialog (*Project* > *Objects*).

3.12.12 Events Triggered by a Thin Client Request

When a script is triggered by a thin client request, information on the request is passed to the script.

The meaning of the properties of the object returned by the RScr.GetCurrentScriptDataEx method:

Property	Meaning
SenderName	The name of the logical computer (based on the project) on which the project is running.
StrPar1	Unique session identifier.
StrPar2	Client IP address.
StrPar3	Client version.
StrPar4	Computer (configuration) name.
StrPar5	User name.
StrPar6	Information on Web browser (User-Agent header). Only for Reliance Smart Client.

IntPar1	Request type: Timeout = 0, Connect = 1, Disconnect = 2, User Log-on = 3, User Log-off = 4
IntPar2	Thin client type: Reliance Web Client = 0, Reliance Mobile Client = 1, Reliance Smart Client = 2
IntPar3	Computer (configuration) Id.
IntPar4	User Id.

3.13 RSys Object

The **RSys** object implements methods that can be divided into two groups:

- Methods for miscellaneous runtime environment operations that are not related to any of the other *Reliance-defined objects* (e.g. **RSys.ActivateWindow**, **RSys.CloseWindow**, **RSys.SetMainWindowTitle**, etc.).
- Methods for miscellaneous operations related to the operating system (e.g. **RSys.Now**, **RSys.SetLocalTime**) and its objects, such as directories and files (e.g. **RSys.DirExists**, **RSys.FileExists**, etc.).

Methods:

- **RSys.ActivateWindow** Procedure
- **RSys.CloseWindow** Procedure
- **RSys.ConvertTimeToDST** Function
- **RSys.CopyFile** Function
- **RSys.CreateDir** Function
- **RSys.DateTimeToInt64Time** Function
- **RSys.DeleteFile** Function
- **RSys.DirExists** Function
- **RSys.ExecApp** Procedure
- **RSys.ExitRuntimeModule** Procedure
- **RSys.FileExists** Function
- **RSys.GetComputerName** Function
- **RSys.GetProjectDir** Function
- **RSys.Int64TimeToDateTime** Function
- **RSys.LocalDateTimeToUTCDateTime** Function
- **RSys.LogMessage** Procedure
- **RSys.Now** Function
- **RSys.PlaySound** Procedure
- **RSys.PrintCustomReport** Procedure

- **RSys.RelativePathToPath** Function
- **RSys.PathToRelativePath** Function
- **RSys.PrintDbReport** Procedure
- **RSys.PrintDbTrend** Procedure
- **RSys.PrintTagDbTrend** Procedure
- **RSys.RemoveDir** Function
- **RSys.RenameFile** Function
- **RSys.ReplaceCZChars** Function
- **RSys.RestartProject** Procedure
- **RSys.RestartWindows** Procedure
- **RSys.SaveCustomReport** Procedure
- **RSys.SetLocalTime** Function
- **RSys.SetMainWindowTitle** Procedure
- **RSys.ShowCustomReport** Procedure
- **RSys.ShowDbReport** Procedure
- **RSys.ShowDbTrend** Procedure
- **RSys.ShowTagDbTrend** Procedure
- **RSys.ShutDownWindows** Procedure
- **RSys.SetProgramLanguage** Procedure
- **RSys.SetProjectLanguage** Procedure
- **RSys.Sleep** Procedure
- **RSys.UTCDateTimeToLocalDateTime** Function

3.13.1 RSys.ActivateWindow Procedure

RSys.ActivateWindow activates a project window.

Syntax

RSys.ActivateWindow *Window*: **Variant**

Argument	Description
Window	The name or ID of the window.

 **RSys** Object

Example

```
RSys.ActivateWindow "MainWindow" ' Activate the window MainWindow.
```

3.13.2 RSys.CloseWindow Procedure

RSys.CloseWindow closes a project window.

Syntax

RSys.CloseWindow *Window*: **Variant**

Argument	Description
Window	The name or ID of the window.

Remarks

The specified window must be a *Dialog* window.

 **RSys** Object

Example

```
RSys.CloseWindow "Settings" ' Close the window Settings.
```

3.13.3 RSys.ConvertTimeToDST Function

RSys.ConvertTimeToDST converts a specified date and time value to daylight saving time.

Syntax

RSys.ConvertTimeToDST(*Value*: DateTime): DateTime

Argument	Description
Value	The date and time value to be converted.

Return values

The method returns a date and time value converted to daylight saving time.

Remarks

This method can be useful when processing data time-stamped in standard time. In order for the method to work, the start and end time of the daylight saving time period must be configured through the TimeConv.ini file located in a visualization project's root directory. If the specified time stamp is within the period, the return value is equal to the time stamp plus one hour. If the specified time stamp is not within the period or the period has not been specified, the return value is equal to the time stamp.

RSys Object

Example

```
Dim DeviceTime, ComputerTime
' Store the current date and time retrieved from PLC1 in the variable DeviceTime.
DeviceTime = RTag.GetTagValue("PLC1", "Time")
' Convert DeviceTime to daylight saving time and store the result in the variable
ComputerTime.
ComputerTime = RSys.ConvertTimeToDST(DeviceTime)
' Change the current date and time to the value stored in the variable ComputerTime.
If RSys.SetLocalTime(ComputerTime) Then
  ' ...
End If
```

3.13.4 RSys.CopyFile Function

RSys.CopyFile copies a source file to a destination file.

Syntax

RSys.CopyFile(*SourceFile*, *DestFile*: **String;** *FailIfExists*: **Boolean): Boolean**

Argument	Description
SourceFile	The full name of the source file.
DestFile	The full name of the destination file.
FailIfExists	Determines how this operation is to proceed if a file of the same name as that specified by *DestFile* already exists (see the following table).

Value	Meaning
True	The method does nothing and fails.
False	The method tries to overwrite the destination file.

Return values

Value	Meaning
True	The file was copied.
False	The file was not copied.

Remarks

The return value is **False** in these cases:

- The source file does not exist.
- The names of the source and destination files are identical.
- The name of the destination file is not valid, since it does not follow the rules for naming files.

- The destination file already exists and *FailIfExists* = **True**.
- The destination file already exists, *FailIfExists* = **False**, but the runtime software was denied access to the destination file.
- There is not sufficient space available on the destination drive to copy the file.

 **RSys** Object

Example

```
' If the file "C:\Data\Data.txt" exists.
If RSys.FileExists("C:\Data\Data.txt") Then
  ' Copy the file to the directory "C:\Backup"
  ' overwriting any file that might exist with the same name.
  If RSys.CopyFile("C:\Data\Data.txt", "C:\Backup\Data.txt", False) Then
    ' ...
  End If
  ' Delete the file "C:\Data\Data.001".
  RSys.DeleteFile "C:\Data\Data.001"
  ' Rename the file "C:\Data\Data.txt" to "C:\Data\Data.001".
  If RSys.RenameFile("C:\Data\Data.txt", "C:\Data\Data.001") Then
    ' ...
  End If
End If
```

3.13.5 RSys.CreateDir Function

RSys.CreateDir creates a specified directory.

Syntax

RSys.CreateDir(*DirName***: String): Boolean**

Argument	Description
DirName	The full path to the directory.

Return values

Value	Meaning

True	The directory has been created.
False	The directory has not been created.

Remarks

By a single call to this method, you can create an entire directory structure. A trailing backslash at the end of the specified path is accepted, but not required.

The return value is **False** in these cases:

- The specified path is not valid, since it does not follow the rules for naming directories.
- The runtime software was denied access when trying to create the directory.

 **RSys** Object

Example

```
' If the directory "C:\Data" does not exist.
If not RSys.DirExists("C:\Data") Then
  If RSys.CreateDir("C:\Data") Then ' Create the directory.
    ' ...
  End If
End If
```

3.13.6 RSys.DateTimeToInt64Time Function

RSys.DateTimeToInt64Time converts a specified date and time value from the *DateTime* format to the *Int64Time* format (used by **Reliance** as the native date and time format).

Date and time in the *Int64Time* format is a value stored as *Int64*, i.e. a 64-bit integer value (in **Reliance** projects, the *Int64* type is represented by the *LargeInt* type). The *Int64Time* format is based on the *FILETIME* format (well-known from the Windows API) which is used by file systems to store file timestamps. It represents the number of 100-nanosecond intervals since January 1, 1601. **Reliance** uses the *Int64Time* format, for example, to store the timestamps of historical data (if it is logged into an SQL-based database) and alarm/event timestamps.

Syntax

RSys.DateTimeToInt64Time(*Value*: DateTime): Int64

Argument	Description
Value	The date and time value to be converted.

Return values

The method returns a date and time value in the *Int64Time* format.

 **RSys** Object

Example

```
Dim DateTimeValue, Int64TimeValue
' Get the value of the tag DateTimeValue from the device System.
DateTimeValue = RTag.GetTagValue("System", "DateTimeValue")
' Convert the date and time value to the Int64Time format.
Int64TimeValue = RSys.DateTimeToInt64Time(DateTimeValue)
' Store the result in the tag Int64TimeValue from the device System.
RTag.SetTagValue "System", "Int64TimeValue", Int64TimeValue
```

3.13.7 RSys.DeleteFile Function

RSys.DeleteFile deletes a specified file.

Syntax

RSys.DeleteFile(*FileName*: String): Boolean

Argument	Description
FileName	The full name of the file.

Return values

Value	Meaning

True	The file has been deleted.
False	The file has not been deleted.

Remarks

The return value is **False** in these cases:

- The file does not exist.
- The runtime software was denied access to the file.

RSys Object

Example

```
' If the file "C:\Data\Data.txt" exists.
If RSys.FileExists("C:\Data\Data.txt") Then
  ' Copy the file to the directory "C:\Backup"
  ' overwriting any file that might exist with the same name.
  If RSys.CopyFile("C:\Data\Data.txt", "C:\Backup\Data.txt", False) Then
    ' ...
  End If
  ' Delete the file "C:\Data\Data.001".
  RSys.DeleteFile "C:\Data\Data.001"
  ' Rename the file "C:\Data\Data.txt" to "C:\Data\Data.001".
  If RSys.RenameFile("C:\Data\Data.txt", "C:\Data\Data.001") Then
    ' ...
  End If
End If
```

3.13.8 RSys.DirExists Function

RSys.DirExists determines whether a specified directory exists.

Syntax

RSys.DirExists(*DirName***: String): Boolean**

Argument	Description
DirName	The full path to the directory.

Return values

Value	Meaning
True	The directory exists.
False	The directory does not exist.

Remarks

A trailing backslash at the end of the specified path is accepted, but not required.

 **RSys** Object

Example

```
' If the directory "C:\Data" does not exist.
If not RSys.DirExists("C:\Data") Then
  If RSys.CreateDir("C:\Data") Then ' Create the directory.
    ' ...
  End If
End If
```

3.13.9 RSys.ExecApp Procedure

RSys.ExecApp runs a specified application or an application associated with the extension of a specified filename.

Syntax

RSys.ExecApp *FileName*, *Params*: **String**

Argument	Description
FileName	The full name of the application's executable file or of the file to be opened by an application associated with the extension of the name.
Params	The parameters to be passed to the application.

Remarks

The *Params* argument can be an empty string if no parameters are to be passed to the application.

RSys Object

Example

```
' Open the file "C:\Data\Data.txt" by the program
' "C:\WinNT\System32\Notepad.exe".
RSys.ExecApp "C:\WinNT\System32\Notepad.exe", "C:\Data\Data.txt"
' Open the file "C:\Data\Data.txt" by the program
' associated with the .txt extension (e.g. Notepad).
RSys.ExecApp "C:\Data\Data.txt", ""
' Open the file "C:\Docs\Data.doc" by the program
' associated with the .doc extension (e.g. Microsoft Word).
RSys.ExecApp "C:\Docs\Data.doc", ""
```

3.13.10 RSys.ExitRuntimeModule Procedure

RSys.ExitRuntimeModule terminates the runtime software.

Syntax

RSys.ExitRuntimeModule

Remarks

The method does not check if the user currently logged on to the runtime software (if any) has sufficient*Access rights* for terminating a visualization project. If it is desired to check the access rights, use methods of the RUser object.

RSys Object

Example

```
Dim UserName
' If there is a user logged on to the runtime software.
If RUser.GetLoggedOnUserName(UserName) Then
  ' If the user has the Servicing access right.
  If RUser.CheckUserAccessRights(UserName, "S") Then
    RSys.ShutDownWindows   ' Shut down the operating system.
```

```
  Else
    RSys.ExitRuntimeModule ' Otherwise terminate the runtime software.
  End If
End If
```

3.13.11 RSys.FileExists Function

RSys.FileExists determines whether a specified file exists.

Syntax

RSys.FileExists(*FileName*: String): Boolean

Argument	Description
FileName	The full name of the file.

Return values

Value	Meaning
True	The file exists.
False	The file does not exist.

RSys Object

Example

```
' If the file "C:\Data\Data.txt" exists.
If RSys.FileExists("C:\Data\Data.txt") Then
  ' Copy the file to the directory "C:\Backup"
  ' overwriting any file that might exist with the same name.
  If RSys.CopyFile("C:\Data\Data.txt", "C:\Backup\Data.txt", False) Then
    ' ...
  End If
  ' Delete the file "C:\Data\Data.001".
  RSys.DeleteFile "C:\Data\Data.001"
  ' Rename the file "C:\Data\Data.txt" to "C:\Data\Data.001".
  If RSys.RenameFile("C:\Data\Data.txt", "C:\Data\Data.001") Then
    ' ...
  End If
End If
```

3.13.12 RSys.GetComputerName Function

RSys.GetComputerName returns the name of a logical computer, defined in a visualization project, on which the project is running.

Syntax

RSys.GetComputerName: String

Return values

The method returns the name of a logical computer, defined in a visualization project, on which the project is running.

 **RSys** Object

Example

```
' If a visualization project is running on the computer PC1.
If RSys.GetComputerName = "PC1" Then
  ' Disable the script Script1.
  RScr.DisableScript "Script1"
End If
```

3.13.13 RSys.GetProjectDir Function

RSys.GetProjectDir returns the full path to a directory where the current visualization project is located.

Syntax

RSys.GetProjectDir: String

Return values

The method returns the full path to the directory where the current visualization project is located.

Remarks

The returned path always contains a trailing backslash.

RSys Object

Example

```
Dim PrjDir
PrjDir = RSys.GetProjectDir
' Run the application "Reports.exe" located
' in a visualization project's Apps directory.
RSys.ExecApp PrjDir + "Main\Apps\Reports.exe", ""
```

3.13.14 RSys.Int64TimeToDateTime Function

RSys.Int64TimeToDateTime converts a specified date and time value from the *Int64Time* format (used by **Reliance** as the native date and time format) to the *DateTime* format.

Date and time in the *Int64Time* format is a value stored as *Int64*, i.e. a 64-bit integer value (in **Reliance** projects, the *Int64* type is represented by the *LargeInt* type). The *Int64Time* format is based on the *FILETIME* format (well-known from the Windows API) which is used by file systems to store file timestamps. It represents the number of 100-nanosecond intervals since January 1, 1601. **Reliance** uses the *Int64Time* format, for example, to store the timestamps of historical data (if it is logged into an SQL-based database) and alarm/event timestamps.

Syntax

RSys.Int64TimeToDateTime(*Value***: Int64): DateTime**

Argument	Description
Value	The date and time value to be converted.

Return values

The method returns a date and time value in the *DateTime* format.

Remarks

This method can be useful e.g. when processing historical data logged by **Reliance** into a table in Microsoft SQL Server. For example, you could export certain records from the table to a file in the *CSV* format.

RSys Object

Example

```
Dim DateTimeValue, Int64TimeValue
' Get the value of the tag Int64TimeValue from the device System.
Int64TimeValue = RTag.GetTagValue("System", "Int64TimeValue")
' Convert the date and time value to the DateTime format.
DateTimeValue = RSys.Int64TimeToDateTime(Int64TimeValue)
' Store the result in the tag DateTimeValue from the device System.
RTag.SetTagValue "System", "DateTimeValue", DateTimeValue
```

3.13.15 RSys.LocalDateTimeToUTCDateTime Function

RSys.LocalDateTimeToUTCDateTime converts a specified date and time value from the local time to UTC.

The local time is dependent on the operating system settings (the time zone, automatically adjusting to daylight saving time).

UTC stands for *Coordinated Universal Time*. It is the primary time standard by which the world regulates clocks and time. The UTC time is based on atomic clocks and is very close to Greenwich Mean Time (GMT). Time zones around the world are expressed as positive or negative offsets from UTC.

Syntax

RSys.LocalDateTimeToUTCDateTime(*Value*: DateTime): DateTime

Argument	Description
Value	The date and time value to be converted.

Return values

The method returns a date and time value in the *DateTime* format.

RSys Object

Example

```
Dim UTCDateTimeValue, LocalDateTimeValue
' Get the value of the tag LocalDateTimeValue from the device System.
LocalDateTimeValue = RTag.GetTagValue("System", "LocalDateTimeValue")
```

```
' Convert the date and time value to UTC.
UTCDateTimeValue = RSys.LocalDateTimeToUTCDateTime(LocalDateTimeValue)
' Store the result in the tag UTCDateTimeValue from the device System.
RTag.SetTagValue "System", "UTCDateTimeValue", UTCDateTimeValue
```

3.13.16 RSys.LogMessage Procedure

Logs a defined text to a log file of a runtime software.

Syntaxe

RSys.LogMessage *Text*: **String**

Argument	Description
Text	Defines a text that should be logged to a log file.

Remarks

All log files are stored in the `<Reliance4>\Logs` directory. The filename is based on the name of a runtime program executable and on a date. For example, a log file of the *Reliance 4 Control Server* program that was created on 2009-12-1 will be `R_CtlSrv_2009_12_01.log`.

 Object **RSys**

Example

```
dim Counter


Counter = RTag.GetTagValue("System", "Counter")

RSys.LogMessage "System/Counter = " + CStr(Counter)
```

3.13.17 RSys.Now Function

RSys.Now returns the current local date and time.

Syntax

RSys.Now: DateTime

Return values

The method returns the current local date and time.

Remarks

The return value is the current system date and time, expressed in Coordinated Universal Time (UTC) format, converted to the currently active time zone's corresponding local date and time.

RSys Object

Example

```
Dim NewTime
' Store the current date and time incremented by 1 hour in a variable.
NewTime = RSys.Now + TimeSerial(1, 0, 0)
' Change the current date and time to the value stored in the variable.
If RSys.SetLocalTime(NewTime) Then
  ' ...
End If
```

3.13.18 RSys.PlaySound Procedure

RSys.PlaySound plays a sound stored in a specified file.

Syntax

RSys.PlaySound *FileName*: **String**

Argument	Description
FileName	The name of the file that contains the sound to be played.

Remarks

The file containing the sound must be located in a visualization project's MMedia directory (the *FileName* argument must not contain a path) or in its subdirectory (the *FileName* argument must contain a relative path to the subdirectory).

RSys Object

Example

```
Dim ID
' If the value of the tag WaterTemperature
' from the device PLC1 is greater than 90.
If RTag.GetTagValue("PLC1", "WaterTemperature") > 90 Then
  ' Play the sound "Beep.wav" located
  ' in a visualization project's MMedia directory.
  RSys.PlaySound "Beep.wav"
  ' Send a SMS message informing the recipient of the event.
  If RModem.GSMSendSMSEx("+420123456789", "Water temperature has exceeded the upper
limit.", ID) Then
    ' ...
  End If
End If
```

3.13.19 RSys.PrintCustomReport Procedure

RSys.PrintCustomReport prints a custom report defined in a visualization project to the default printer.

Syntax

RSys.PrintCustomReport *Report*: **Variant**

Argument	Description
Report	The name or ID of the custom report.

RSys Object

Example

```
' Print the custom report "Report1".
RSys.PrintCustomReport "Report1"
```

3.13.20 RSys.PrintDbReport Procedure

RSys.PrintDbReport prints a historical report defined in a visualization project to the default printer.

Syntax

RSys.PrintDbReport *Report*: **Variant**

Argument	Description
Report	The name or ID of the historical report.

RSys Object

Example

```
' Print the historical report "Report1".
RSys.PrintDbReport "Report1"
```

3.13.21 RSys.PrintDbTrend Procedure

RSys.PrintDbTrend prints a historical trend defined in a visualization project to the default printer.

Syntax

RSys.PrintDbTrend *Trend*: **Variant**

Argument	Description
Trend	The name or ID of the historical trend.

RSys Object

Example

```
' Print the historical trend "Trend1".
RSys.PrintDbTrend "Trend1"
```

3.13.22 RSys.PrintTagDbTrend Procedure

RSys.PrintTagDbTrend prints the historical trend of a specified tag to the default printer. The trend is not one the trends defined through the Trend Manager. If the tag's data is logged into multiple data tables, the trend data is loaded from the table that has the shortest sampling interval.

Syntax

RSys.PrintTagDbTrend *DevName*, *TagName*: **String**

Argument	Description
DevName	The name of the device that the tag belongs to.
TagName	The name of the tag.

RSys Object

Example

```
' Print the trend of the tag WaterTemperature from the device PLC1.
RSys.PrintTagDbTrend "PLC1", "WaterTemperature"
```

3.13.23 RSys.PathToRelativePath Function

Converts a specified absolute path to the corresponding relative path.

Syntaxe

RSys.PathToRelativePath(*Path:***String) String**

Argument	Description
Path	Absolute path to be converted.

Remarks

In the context of **Reliance**, a *relative path* means a directory path defined using an *environment variable*, e.g. the path `$(Reliance)\Utils\`. Available environment variables are listed in the following table.

Reliance environment variables

Variable	Value
$(Reliance)\	<Reliance4>\ directory, i.e. the directory of Reliance 4 program files
$(Components)\	<Reliance4>\Components\ directory, i.e. the directory of Reliance 4 components (graphical objects)
$(Drivers)\	<Reliance4>\Drivers\ directory, i.e. the directory of Reliance 4 communication drivers
$(Project)\	<Project>\ directory, i.e. the directory of the current visualization project
$(CustomReports)\	<Project>\Main\CustomReports\ directory
$(SettingsProfiles)\	<Project>\Settings\Profiles\ directory
$(SettingsComponents)\	<Project>\Settings\Components\ directory
$(SettingsRecipes)\	<Project>\Settings\Recipes\ directory
$(HistoryAlarmsEvents)\	<Project>\History\AlarmsEvents\ directory
$(HistoryData)\	<Project>\History\Data\ directory
$(HistoryPostmort)\	<Project>\History\Postmort\ directory
$(HistoryWindowRecords)\	<Project>\History\WindowRecords\ directory
$(UserDocuments)\	%USERPROFILE%\Dokumenty\ directory, i.e. the user data directory
$(ApplicationData)\	%PROGRAMDATA%\ directory, i.e. the program data directory

Object **RSys**

Example

```
' Convert a path to the corresponding relative path and store it in a tag named
"RelativePath"
RTag.SetTagValue "System", "RelativePath", RSys.PathToRelativePath(RTag.GetTagValue(
"System", "Path"))
```

3.13.24 RSys.RelativePathToPath Function

Converts a specified relative path to the corresponding relative path.

Syntaxe

RSys.RelativePathToPath(*RelativePath:***String) String**

Argument	Description
RelativePath	Relative path to be converted.

Remarks

In the context of **Reliance**, a *relative path* means a directory path defined using an *environment variable*, e.g. the path `$(Reliance)\Utils\`. Available environment variables are listed in the following table.

Reliance environment variables

Variable	Value
$(Reliance)\	<Reliance4>\ directory, i.e. the directory of Reliance 4 program files
$(Components)\	<Reliance4>\Components\ directory, i.e. the directory of Reliance 4 components (graphical objects)
$(Drivers)\	<Reliance4>\Drivers\ directory, i.e. the directory of Reliance 4 communication drivers
$(Project)\	<Project>\ directory, i.e. the directory of the current visualization project
$(CustomReports)\	<Project>\Main\CustomReports\ directory
$(SettingsProfiles)\	<Project>\Settings\Profiles\ directory
$(SettingsComponents)\	<Project>\Settings\Components\ directory
$(SettingsRecipes)\	<Project>\Settings\Recipes\ directory
$(HistoryAlarmsEvents)\	<Project>\History\AlarmsEvents\ directory
$(HistoryData)\	<Project>\History\Data\ directory

$(HistoryPostmort)\	<Project>\History\Postmort\ directory
$(HistoryWindowRecords)\	<Project>\History\WindowRecords\ directory
$(UserDocuments)\	%USERPROFILE%\Dokumenty\ directory, i.e. the user data directory
$(ApplicationData)\	%PROGRAMDATA%\ directory, i.e. the program data directory

Object **RSys**

Example

```
' Convert a relative path to the corresponding absolute path and store it in a tag named "Path"
RTag.SetTagValue "System", "Path", RSys.RelativePathToPath(RTag.GetTagValue("System", "RelativePath"))
```

3.13.25 RSys.RemoveDir Function

RSys.RemoveDir removes (i.e. deletes) a specified directory.

Syntax

RSys.RemoveDir(*DirName***: String): Boolean**

Argument	Description
DirName	The full path to the directory.

Return values

Value	Meaning
True	The directory has been removed.
False	The directory has not been removed.

Remarks

A trailing backslash at the end of the specified path is accepted, but not required.

The return value is **False** in these cases:

- The directory does not exist.
- The directory is not empty.
- The runtime software was denied access when trying to remove the directory.

 **RSys** Object

Example

```
If RSys.RemoveDir("C:\Data") Then
  ' ...
End If
```

3.13.26 RSys.RenameFile Function

RSys.RenameFile renames a specified file.

Syntax

RSys.RenameFile(*OldName*, *NewName*: **String): Boolean**

Argument	Description
OldName	The original full name of the file.
NewName	The new full name for the file.

Return values

Value	Meaning
True	The file has been renamed.
False	The file has not been renamed.

Remarks

The return value is **False** in these cases:

- The original file does not exist.

- The path to the new file does not exist.
- The runtime software was denied access to the file.

RSys Object

Example

```
' If the file "C:\Data\Data.txt" exists.
If RSys.FileExists("C:\Data\Data.txt") Then
  ' Copy the file to the directory "C:\Backup"
  ' overwriting any file that might exist with the same name.
  If RSys.CopyFile("C:\Data\Data.txt", "C:\Backup\Data.txt", False) Then
    ' ...
  End If
  ' Delete the file "C:\Data\Data.001".
  RSys.DeleteFile "C:\Data\Data.001"
  ' Rename the file "C:\Data\Data.txt" to "C:\Data\Data.001".
  If RSys.RenameFile("C:\Data\Data.txt", "C:\Data\Data.001") Then
    ' ...
  End If
End If
```

3.13.27 RSys.ReplaceCZChars Function

RSys.ReplaceCZChars converts a specified text string by replacing characters containing Czech diacritical marks with corresponding characters of the English alphabet and returns the resulting string.

Syntax

RSys.ReplaceCZChars(*Text***: String): String**

Argument	Description
Text	The text string to be converted.

Return values

The method returns a text string with no characters containing Czech diacritical marks.

Remarks

This method can be useful when sending SMS messages through a call to the RModem.GSMSendSMS and RModem.GSMSendSMSEx methods, which do not support characters containing diacritical marks.

RSys Object

Example

```
Dim ID
' If the value of the tag WaterTemperature
' from the device PLC1 is greater than 90.
If RTag.GetTagValue("PLC1", "WaterTemperature") > 90 Then
  ' Play the sound "Beep.wav" located
  ' in a visualization project's MMedia directory.
  RSys.PlaySound "Beep.wav"
  ' Send a SMS message informing the recipient of the event.
  ' The recipient should receive the following Czech message:
  ' Teplota vody prekrocila horni mez.
  If RModem.GSMSendSMSEx("+420123456789", RSys.ReplaceCZChars("Teplota vody překročila
horní mez."), ID) Then
    ' ...
  End If
End If
```

3.13.28 RSys.RestartProject Procedure

Terminates the project and starts it again using the settings defined for a specified computer. The runtime software is not terminated during this operation (it keeps running).

Syntax

RSys.RestartProject *Computer*: **Variant**

Argument	Description
Computer	The name or ID (as defined in the project) of the computer whose settings should be used when starting the project. A special value of "" means the computer on which the project is currently running (whose settings are currently being used).

Remarks

Restarting the project (on the same computer) can be useful, for example, to perform the automatic project update. The update is performed while the project starts if the respective option is active.

Terminating the project and starting it again using the settings defined for another computer can be used, for example, when a different way of communication to I/O devices should be used (e.g. directly with communication drivers instead of communication through a data server).

The method does not check if the user currently logged on to the runtime software (if any) has sufficient access rights for terminating the project. If it is desired to check the access rights, use methods of the RUser object.

RSys Object

Example

```
Dim UserName
' If there is a user logged on to the runtime software.
If RUser.GetLoggedOnUserName(UserName) Then
  ' If the user has the Servicing access right.
  If RUser.CheckUserAccessRights(UserName, "S") Then
    RSys.RestartProject ""    ' Restarts the project.
  End If
End If
```

3.13.29 RSys.RestartWindows Procedure

RSys.RestartWindows terminates the runtime software and restarts the operating system.

Syntax

RSys.RestartWindows

Remarks

The method does not check if the user currently logged on to the runtime software (if any) has sufficient access rights for terminating the project. If it is desired to check the access rights, use methods of the RUser object.

RSys Object

Example

```
Dim UserName
' If there is a user logged on to the runtime software.
If RUser.GetLoggedOnUserName(UserName) Then
  ' If the user has the Servicing access right.
  If RUser.CheckUserAccessRights(UserName, "S") Then
    RSys.RestartWindows    ' Restarts the operating system.
  End If
End If
```

3.13.30 RSys.SaveCustomReport Procedure

RSys.SaveCustomReport saves a custom report defined in a visualization project to a specified file.

Syntax

RSys.SaveCustomReport *Report*: **Variant;** *FileName*: **String**

Argument	Description
Report	The name or ID of the custom report.
FileName	The file to which the custom report is to be saved.

Remarks

In case of custom reports of type *FastReport*, the file extension determines the format of the created document.

Document formats

Document format	Extension
Portable Document Format	pdf
Excel 97-2003 sheet	xls
Excel sheet	xlsx
Data files in XML format	xml
CSV (delimited by semicolon)	csv
Web page	htm or html
FastReport	rrp or fr3

 **RSys** Object

Example

```
' Save the report "Report1" to a specified file.
RSys.SaveCustomReport "Report1", "C:\Reliance\CustomReports\Report1.htm"
```

3.13.31 RSys.SetLocalTime Function

RSys.SetLocalTime sets the current local date and time to a specified value.

Syntax

RSys.SetLocalTime(*Value*: DateTime): Boolean

Argument	Description
Value	The new value for the current local date and time.

Return values

Value	Meaning
True	The date and time has been set to the new value.
False	The date and time has not been set to the new value.

Remarks

In order for the call to succeed, the user currently logged on to the operating system must have the appropriate security privilege for this operation.

 **RSys** Object

Example

```
Dim NewTime
' Store the current date and time incremented by 1 hour in a variable.
NewTime = RSys.Now + TimeSerial(1, 0, 0)
' Change the current date and time to the value stored in the variable.
If RSys.SetLocalTime(NewTime) Then
  ' ...
End If
```

3.13.32 RSys.SetMainWindowTitle Procedure

RSys.SetMainWindowTitle changes the title of the runtime software's main window by appending a specified text to the default title (e.g. *Reliance Control*).

Syntax

RSys.SetMainWindowTitle *Title*: **String**

Argument	Description
Title	The text to be appended to the default title.

 **RSys** Object

Example

```
' After the call, the title would be: Reliance Control - Demo.
RSys.SetMainWindowTitle "Demo"
```

3.13.33 RSys.ShowCustomReport Procedure

RSys.ShowCustomReport shows a custom report defined in a visualization project.

Syntax

RSys.ShowCustomReport *Report*: **Variant;** *AsStandaloneWindow*: **Boolean**

Argument	Description
Report	The name or ID of the custom report.
AsStandaloneWindow	Determines whether to show the custom report in a stand-alone window.

 **RSys** Object

Example

```
' Show the report "Report1" in a stand-alone window.
```

```
RSys.ShowCustomReport "Report1", True
```

3.13.34 RSys.ShowDbReport Procedure

RSys.ShowDbReport shows a historical report defined in a visualization project.

Syntax

RSys.ShowDbReport *Report*: **Variant;** *AsStandaloneWindow*: **Boolean**

Argument	Description
Report	The name or ID of the report.
AsStandaloneWindow	Determines whether to show the report in a stand-alone window.

RSys Object

Example

```
' Show the report "Report1" in a stand-alone window.
RSys.ShowDbReport "Report1", True
```

3.13.35 RSys.ShowDbTrend Procedure

RSys.ShowDbTrend shows a historical trend defined in a visualization project.

Syntax

RSys.ShowDbTrend *Trend*: **Variant;** *AsStandaloneWindow*: **Boolean**

Argument	Description
Trend	The name or ID of the trend.
AsStandaloneWindow	Determines whether to show the trend in a stand-alone window.

RSys Object

Example

```
' Show the trend "Trend1" in a stand-alone window.
RSys.ShowDbTrend "Trend1", True
```

3.13.36 RSys.ShowTagDbTrend Procedure

RSys.ShowTagDbTrend shows the historical trend of a specified tag. The trend is not one the trends defined through the Trend Manager. If the tag's data is logged into multiple data tables, the trend data is loaded from the table that has the shortest sampling interval.

Syntax

RSys.ShowTagDbTrend *DevName*, *TagName*: **String;** *AsStandaloneWindow*: **Boolean**

Argument	Description
DevName	The name of the device that the tag belongs to.
TagName	The name of the tag.
AsStandaloneWindow	Determines whether to show the trend in a stand-alone window.

RSys Object

Example

```
' Show the trend of the tag WaterTemperature from the device PLC1 in a stand-alone
window.
RSys.ShowTagDbTrend "PLC1", "WaterTemperature", True
```

3.13.37 RSys.ShutDownWindows Procedure

RSys.ShutDownWindows terminates the runtime software and shuts down the operating system.

Syntax

RSys.ShutDownWindows

RSys Object

Example

```
Dim UserName
' If there is a user logged on to the runtime software.
If RUser.GetLoggedOnUserName(UserName) Then
  ' If the user has the Servicing access right.
  If RUser.CheckUserAccessRights(UserName, "S") Then
    RSys.ShutDownWindows   ' Shut down the operating system.
  Else
    RSys.ExitRuntimeModule ' Otherwise terminate the runtime software.
  End If
End If
```

3.13.38 RSys.SetProgramLanguage Procedure

RSys.SetProgramLanguage sets the runtime software's user interface language to a specified language.

Syntax

RSys.SetProgramLanguage *Language*: **Variant**

Argument	Description
Language	The abbreviation or index of the language.

List of language abbreviations:

Language	Index	Abbreviation
Czech	*0*	CSY
English	*1*	ENU
Polish	*2*	PLK
Russian	*3*	RUS
German	*4*	DEU
Lithuanian	*5*	LTH

Hungarian	*6*	HUN
Slovak	*8*	SKY
Greek	*9*	ELL

RSys Object

Example

```
RSys.SetProgramLanguage "CSY" ' sets the runtime software's user interface language to
Czech language.
```

3.13.39 RSys.SetProjectLanguage Procedure

RSys.SetProjectLanguage sets the project's active language to a specified language.

Syntax

RSys.SetProjectLanguage *Language*: **Variant**

Argument	Description
Language	The name or ID of the language.

RSys Object

Example

```
RSys.SetProjectLanguage "Czech (Czech Republic)" ' Sets the project's active language
to "Czech (Czech Republic)".
```

3.13.40 RSys.Sleep Procedure

Suspends the execution of the current script for a specified time.

Syntaxe

RSys.Sleep *Interval*: **Integer**

Argument	Description
Interval	Time interval (ms).

 Object **RSys**

Example

```
' Suspend the current script for 2 seconds
RSys.Sleep 2000
```

3.13.41 RSys.UTCDateTimeToLocalDateTime Function

RSys.UTCDateTimeToLocalDateTime converts a specified date and time value from UTC to the local time.

UTC stands for *Coordinated Universal Time*. It is the primary time standard by which the world regulates clocks and time. The UTC time is based on atomic clocks and is very close to Greenwich Mean Time (GMT). Time zones around the world are expressed as positive or negative offsets from UTC.

The local time is dependent on the operating system settings (the time zone, automatically adjusting to daylight saving time).

Syntax

RSys.UTCDateTimeToLocalDateTime(*Value***: DateTime): DateTime**

Argument	Description
Value	The date and time value to be converted.

Return values

The method returns a date and time value in the *DateTime* format.

Remarks

This method can be useful e.g. when processing historical data logged by **Reliance** into a table in Microsoft SQL Server if the record timestamps are stored in UTC. For example, you could export certain records from the table to a file in the *CSV* format with timestamps in the local time.

RSys Object

Example

```
Dim LocalDateTimeValue, UTCDateTimeValue
' Get the value of the tag UTCDateTimeValue from the device System.
UTCDateTimeValue = RTag.GetTagValue("System", "UTCDateTimeValue")
' Convert the date and time value to the local time.
LocalDateTimeValue = RSys.UTCDateTimeToLocalDateTime(UTCDateTimeValue)
' Store the result in the tag LocalDateTimeValue from the device System.
RTag.SetTagValue "System", "LocalDateTimeValue", LocalDateTimeValue
```

3.14 TTable-type Objects

TTable-type objects implement methods and properties for operations on database tables, either current or archive, belonging to databases defined in a visualization project. To create a new **TTable**-type object, use the RDb.CreateTableObject method.

Properties:

- **TTable.ArchiveName** Property
- **TTable.DatabaseName** Property
- **TTable.DateFieldValue** Property
- **TTable.IsArchive** Property
- **TTable.TimeFieldValue** Property

Methods:

- **TTable.Append** Procedure
- **TTable.Bof** Function
- **TTable.Cancel** Procedure
- **TTable.CloseTable** Procedure
- **TTable.CreateTable** Function
- **TTable.Delete** Procedure
- **TTable.DeleteTable** Function
- **TTable.Edit** Procedure
- **TTable.EmptyTable** Function
- **TTable.Eof** Function
- **TTable.FieldExists** Function
- **TTable.First** Procedure
- **TTable.GetFieldValue** Function
- **TTable.Last** Procedure
- **TTable.MoveBy** Procedure
- **TTable.Next** Procedure

- **TTable.OpenTable** Function
- **TTable.Post** Procedure
- **TTable.Prior** Procedure
- **TTable.SetFieldValue** Procedure
- **TTable.TableExists** Function
- **TTable.UpdateTableStructure** Procedure

3.14.1 TTable.ArchiveName Property

TTable.ArchiveName determines the full name of the archive database table to be accessed by this object.

Syntax

TTable.ArchiveName: String

Remarks

The property is read-write. It only makes sense if TTable.IsArchive = **True**.

- **TTable**-type Objects

Example

```
Dim Table
Set Table = RDb.CreateTableObject ' Create a TTable-type object.
' Name of the database, as defined in a visualization project.
Table.DatabaseName = "Database1"
' We want to access an archive table that contains data from August, 2000.
Table.IsArchive = True
Table.ArchiveName = "C:\Reliance\Projects\Test\Data\2000\d1_0008.DB"
If Table.TableExists Then          ' If the archive table exists.
  If Table.OpenTable Then          ' If the table can be opened.
    ' ...
    Table.CloseTable               ' Close the table.
  End If
End If
Set Table = Nothing                ' Free the TTable-type object.
```

3.14.2 TTable.DatabaseName Property

TTable.DatabaseName determines the data table to associate with this object. It corresponds to the *name* property of the data table defined via the *Data Table Manager.*

Syntax

TTable.DatabaseName: String

Remarks

The property is read-write.

TTable-type Objects

Example

```
Dim Table
Set Table = RDb.CreateTableObject ' Create a TTable-type object.
' Name of the data table, as defined in a visualization project.
Table.DatabaseName = "Water"
If Table.OpenTable Then           ' If the current table can be opened.
  Table.Append                    ' Append a new record.
  ' Save current system date to the new record.
  Table.DateFieldValue = Date
  ' Save current system time to the new record.
  Table.TimeFieldValue = Time
  ' Save the value of the tag WaterTemperature from the device PLC1 to the new record.
  Table.SetFieldValue "PLC1", "WaterTemperature", RTag.GetTagValue("PLC1",
"WaterTemperature")
  Table.Post                      ' Write the new record to the table.
  Table.CloseTable                ' Close the table.
End If
Set Table = Nothing               ' Free the TTable-type object.
```

3.14.3 TTable.DateFieldValue Property

TTable.DateFieldValue determines the value of the date field of a database table's active record.

Syntax

TTable.DateFieldValue: DateTime

Remarks

The property is read-write. The table must be open and not empty. In addition, the table must be in edit or insert mode before you can write the property. Every table belonging to a database defined in a visualization project contains a field for storing a record's date which is returned by this property.

TTable-type Objects

Example

```
Dim Table
Set Table = RDb.CreateTableObject ' Create a TTable-type object.
' Name of the database, as defined in a visualization project.
Table.DatabaseName = "Water"
If Table.OpenTable Then           ' If the current table can be opened.
  Table.Append                    ' Append a new record.
  ' Save current system date to the new record.
  Table.DateFieldValue = Date
  ' Save current system time to the new record.
  Table.TimeFieldValue = Time
  ' Save the value of the tag WaterTemperature from the device PLC1 to the new record.
  Table.SetFieldValue "PLC1", "WaterTemperature", RTag.GetTagValue("PLC1",
"WaterTemperature")
  Table.Post                      ' Write the new record to the table.
  Table.CloseTable                ' Close the table.
End If
Set Table = Nothing               ' Free the TTable-type object.
```

3.14.4 TTable.IsArchive Property

TTable.IsArchive determines whether the object is to access the current or an archive database table.

Syntax

TTable.IsArchive: Boolean

Remarks

The property is read-write.

Value	Meaning
True	The object is to access an archive table (specified by the TTable.ArchiveName property).
False	The object is to access the current table (the default).

 **TTable**-type Objects

Example

```
Dim Table
Set Table = RDb.CreateTableObject ' Create a TTable-type object.
' Name of the database, as defined in a visualization project.
Table.DatabaseName = "Database1"
' We want to access an archive table that contains data from August, 2000.
Table.IsArchive = True
Table.ArchiveName = "C:\Reliance\Projects\Test\Data\2000\d1_0008.DB"
If Table.TableExists Then          ' If the archive table exists.
  If Table.OpenTable Then          ' If the table can be opened.
    ' ...
    Table.CloseTable               ' Close the table.
  End If
End If
Set Table = Nothing                ' Free the TTable-type object.
```

3.14.5 TTable.TimeFieldValue Property

TTable.DateFieldValue determines the value of the time field of a database table's active record.

Syntax

TTable.TimeFieldValue: DateTime

Remarks

The property is read-write. The table must be open and not empty. In addition, the table must be in edit or insert mode before you can write the property. Every table belonging to a database defined in a visualization project contains a field for storing a record's time which is returned by this property.

 TTable-type Objects

Example

```
Dim Table
Set Table = RDb.CreateTableObject ' Create a TTable-type object.
' Name of the database, as defined in a visualization project.
Table.DatabaseName = "Water"
If Table.OpenTable Then           ' If the current table can be opened.
  Table.Append                    ' Append a new record.
  ' Save current system date to the new record.
  Table.DateFieldValue = Date
  ' Save current system time to the new record.
  Table.TimeFieldValue = Time
  ' Save the value of the tag WaterTemperature from the device PLC1 to the new record.
  Table.SetFieldValue "PLC1", "WaterTemperature", RTag.GetTagValue("PLC1",
"WaterTemperature")
  Table.Post                      ' Write the new record to the table.
  Table.CloseTable                ' Close the table.
End If
Set Table = Nothing               ' Free the TTable-type object.
```

3.14.6 TTable.Append Procedure

TTable.Append adds a new, empty record at the end of a database table and makes it the active record.

Syntax

TTable.Append

Remarks

The table must be open. After a call to this method, the table is in insert mode. In insert mode, it is possible to modify the new record by setting the values of database fields. After modifying the record, call the TTable.Post method to post the record (i.e. write it to the table). Setting the active record to another record using the TTable.First, TTable.Last, TTable.Next, TTable.Prior, TTable.MoveBy methods also posts the record. If the record has not yet been posted, it can be canceled by calling the TTable.Cancel method or closing the table using the TTable.CloseTable method.

TTable-type Objects

Example

```
Dim Table
Set Table = RDb.CreateTableObject ' Create a TTable-type object.
' Name of the database, as defined in a visualization project.
Table.DatabaseName = "Water"
If Table.OpenTable Then           ' If the current table can be opened.
  Table.Append                    ' Append a new record.
  ' Save current system date to the new record.
  Table.DateFieldValue = Date
  ' Save current system time to the new record.
  Table.TimeFieldValue = Time
  ' Save the value of the tag WaterTemperature from the device PLC1 to the new record.
  Table.SetFieldValue "PLC1", "WaterTemperature", RTag.GetTagValue("PLC1",
"WaterTemperature")
  Table.Post                      ' Write the new record to the table.
  Table.CloseTable                ' Close the table.
End If
Set Table = Nothing               ' Free the TTable-type object.
```

3.14.7 TTable.Bof Function

TTable.Bof indicates whether a database cursor is positioned at the beginning of a database table (Bof stands for Beginning of file).

Syntax

TTable.Bof: Boolean

Return values

Value	Meaning
True	The cursor is guaranteed to be positioned at the beginning of the table.
False	The cursor is not guaranteed to be positioned at the beginning of the table.

Remarks

The table must be open. This method is useful in combination with the TTable.First, TTable.Last, TTable.Next, TTable.Prior, TTable.MoveBy, TTable.Eof methods.

TTable.Bof returns **True** when a script:

- Opens the table.
- Calls the TTable.First method.
- Calls the TTable.Prior method, and the call fails (because the cursor is already positioned at the first record in the table).

TTable.Bof returns **False** in all other cases.

TTable-type Objects

Example

```
Dim Table
Set Table = RDb.CreateTableObject ' Create a TTable-type object.
' Name of the database, as defined in a visualization project.
Table.DatabaseName = "Water"
If Table.OpenTable Then           ' If the current table can be opened.
```

```
  Table.Last                        ' Move to the last record.
  ' While the Table.Bof method returns False.
  While Not Table.Bof
    ' ...
    Table.Prior                     ' Move to the prior record.
  WEnd
  Table.CloseTable                  ' Close the table.
End If
Set Table = Nothing                 ' Free the TTable-type object.
```

3.14.8 TTable.Cancel Procedure

TTable.Cancel cancels changes (if any) to a database table's active record if those changes have not yet been posted (i.e. written to the table).

Syntax

TTable.Cancel

Remarks

The table must be open. Changes can also be canceled by closing the table using the TTable.CloseTable method.

TTable-type Objects

Example

```
Dim Table
Set Table = RDb.CreateTableObject ' Create a TTable-type object.
' Name of the database, as defined in a visualization project.
Table.DatabaseName = "Water"
If Table.OpenTable Then           ' If the current table can be opened.
  Table.Append                    ' Append a new record.
  ' Save current system date to the new record.
  Table.DateFieldValue = Date
  ' Save current system time to the new record.
  Table.TimeFieldValue = Time
  ' Save the value of the tag WaterTemperature from the device PLC1 to the new record.
  Table.SetFieldValue "PLC1", "WaterTemperature", RTag.GetTagValue("PLC1",
"WaterTemperature")
  Table.Cancel                    ' Cancel the new record.
  Table.CloseTable                ' Close the table.
End If
```

```
Set Table = Nothing                ' Free the TTable-type object.
```

3.14.9 TTable.CloseTable Procedure

TTable.CloseTable closes a database table.

Syntax

TTable.CloseTable

Remarks

The method cancels changes (if any) to the table's active record if those changes are not yet written to the table.

TTable-type Objects

Example

```
Dim Table
Set Table = RDb.CreateTableObject ' Create a TTable-type object.
' Name of the database, as defined in a visualization project.
Table.DatabaseName = "Water"
If Table.OpenTable Then            ' If the current table can be opened.
  Table.Append                     ' Append a new record.
  ' Save current system date to the new record.
  Table.DateFieldValue = Date
  ' Save current system time to the new record.
  Table.TimeFieldValue = Time
  ' Save the value of the tag WaterTemperature from the device PLC1 to the new record.
  Table.SetFieldValue "PLC1", "WaterTemperature", RTag.GetTagValue("PLC1",
"WaterTemperature")
  Table.Post                       ' Write the new record to the table.
  Table.CloseTable                 ' Close the table.
End If
Set Table = Nothing                ' Free the TTable-type object.
```

3.14.10 TTable.CreateTable Function

TTable.CreateTable creates a new database table.

Syntax

TTable.CreateTable

Return values

Value	Meaning
True	The table has been created.
False	The table has not been created.

Remarks

The method creates the current or an archive table based on the value of the TTable.IsArchive property. If the table already exists, it is not overwritten and the call fails.

TTable-type Objects

Example

```
Dim Table, TableNotExists
Set Table = RDb.CreateTableObject    ' Create a TTable-type object.
TableNotExists = True
' Name of the database, as defined in a visualization project.
Table.DatabaseName = "Water"
If Table.TableExists Then            ' If the table exists.
  TableNotExists = Table.DeleteTable ' Delete the table.
End If
If TableNotExists Then
  If Table.CreateTable Then          ' Create a new table.
    ' ...
  End If
End If
Set Table = Nothing                  ' Free the TTable-type object.
```

3.14.11 TTable.Delete Procedure

TTable.Delete deletes the active record from a database table and positions a database cursor on the next record.

Syntax

TTable.Delete

Remarks

The table must be open. This operation cannot be reversed. If the table is empty, both the TTable.Bof and TTable.Eof methods return **True**.

 **TTable**-type Objects

Example

```
Dim Table
Set Table = RDb.CreateTableObject       ' Create a TTable-type object.
' Name of the database, as defined in a visualization project.
Table.DatabaseName = "Water"
If Table.OpenTable Then                 ' If the current table can be opened.
  If Not (Table.Bof And Table.Eof) Then ' If the table is not empty.
    Table.Last                          ' Move to the last record.
    Table.Delete                        ' Delete the active record.
  End If
  Table.CloseTable                      ' Close the table.
End If
Set Table = Nothing                     ' Free the TTable-type object.
```

3.14.12 TTable.DeleteTable Function

TTable.DeleteTable deletes an existing database table.

Syntax

TTable.DeleteTable: Boolean

Return values

Value	Meaning

True	The table has been deleted.
False	The table has not been deleted.

Remarks

The table must not be open by another **TTable**-type object or another program (e.g. another instance of the runtime software).

Before calling this method, it is advisable to test whether the table exists by calling the TTable.TableExists method.

 **TTable**-type Objects

Example

```
Dim Table, TableNotExists
Set Table = RDb.CreateTableObject    ' Create a TTable-type object.
TableNotExists = True
' Name of the database, as defined in a visualization project.
Table.DatabaseName = "Water"
If Table.TableExists Then            ' If the table exists.
  TableNotExists = Table.DeleteTable ' Delete the table.
End If
If TableNotExists Then
  If Table.CreateTable Then          ' Create a new table.
    ' ...
  End If
End If
Set Table = Nothing                  ' Free the TTable-type object.
```

3.14.13 TTable.Edit Procedure

TTable.Edit puts a database table into edit mode.

Syntax

TTable.Edit

Remarks

The table must be open. In edit mode, it is possible to modify the active record by setting the values of database fields. After modifying the record, call the TTable.Post method to post the record (i.e. write it to the table). Setting the active record to another record using the TTable.First, TTable.Last, TTable.Next, TTable.Prior, TTable.MoveBy methods also posts the record. If the record has not yet been posted, the changes to the record can be canceled by calling the TTable.Cancel method or closing the table using the TTable.CloseTable method.

TTable-type Objects

Example

```
Dim Table
Set Table = RDb.CreateTableObject ' Create a TTable-type object.
' Name of the database, as defined in a visualization project.
Table.DatabaseName = "Water"
If Table.OpenTable Then           ' If the current table can be opened.
  Table.Last                      ' Move to the last record.
  Table.Edit                      ' Put the table into edit mode.
  ' Save 0 to the database field linked to the tag
  ' WaterTemperature from the device PLC1.
  Table.SetFieldValue "PLC1", "WaterTemperature", 0
  Table.Post                      ' Write the new record to the table.
  Table.CloseTable                ' Close the table.
End If
Set Table = Nothing               ' Free the TTable-type object.
```

3.14.14 TTable.EmptyTable Function

TTable.EmptyTable deletes all records from a database table.

Syntax

TTable.EmptyTable: Boolean

Return values

Value	Meaning
True	Records have been deleted.
False	Records have not been deleted.

Remarks

The table must not be open by another **TTable**-type object or another program (e.g. another instance of the runtime software).

Before calling this method, it is advisable to test whether the table exists by calling the TTable.TableExists method.

TTable-type Objects

Example

```
Dim Table
Set Table = RDb.CreateTableObject ' Create a TTable-type object.
' Name of the database, as defined in a visualization project.
Table.DatabaseName = "Water"
If Table.TableExists Then         ' If the table exists.
  If Table.EmptyTable Then        ' Empty the table.
    ' ...
  End If
End If
Set Table = Nothing               ' Free the TTable-type object.
```

3.14.15 TTable.Eof Function

TTable.Eof indicates whether a database cursor is positioned at the end of a database table (Eof stands for End of file).

Syntax

TTable.Eof: Boolean

Return values

Value	Meaning
True	The cursor is guaranteed to be positioned at the end of the table.
False	The cursor is not guaranteed to be positioned at the end of the table.

Remarks

The table must be open. This method is useful in combination with the TTable.First, TTable.Last, TTable.Next, TTable.Prior, TTable.MoveBy, TTable.Bof methods.

TTable.Eof returns **True** when a script:

- Opens an empty table.
- Calls the TTable.Last method.
- Calls the TTable.Next method, and the call fails (because the cursor is already positioned at the last record in the table).

TTable.Eof returns **False** in all other cases.

 **TTable**-type Objects

Example

```
Dim Table
Set Table = RDb.CreateTableObject ' Create a TTable-type object.
' Name of the database, as defined in a visualization project.
Table.DatabaseName = "Water"
If Table.OpenTable Then           ' If the current table can be opened.
```

```
  Table.First                    ' Move to the first record.
  ' While the Table.Eof method returns False.
  While Not Table.Eof
    ' ...
    Table.Next                   ' Move to the next record.
  WEnd
  Table.CloseTable               ' Close the table.
End If
Set Table = Nothing              ' Free the TTable-type object.
```

3.14.16 TTable.FieldExists Function

TTable.FieldExists indicates whether a field linked to a specified tag exists in a database table.

Syntax

TTable.FieldExists(*DevName*, *TagName*: **String): Boolean**

Argument	Description
DevName	The name of the device that the tag belongs to.
TagName	The name of the tag that the field is linked to.

Return values

Value	Meaning
True	The field exists in the table.
False	The field does not exist in the table.

Remarks

The table must be open.

 **TTable**-type Objects

Example

```
Dim Table
```

```
Set Table = RDb.CreateTableObject ' Create a TTable-type object.
' Name of the database, as defined in a visualization project.
Table.DatabaseName = "Water"
If Table.OpenTable Then           ' If the current table can be opened.
  ' If the database field linked to the tag WaterTemperature
  ' from the device PLC1 does not exist the table has an old structure.
  If not Table.FieldExists("PLC1", "WaterTemperature") Then
    Table.CloseTable              ' Close the table.
    If Table.DeleteTable Then     ' If the table can be deleted.
      If Table.CreateTable Then   ' If a new table can be created.
        If Table.OpenTable Then   ' If the table can be opened.
          ' ...
          Table.CloseTable        ' Close the table.
        End If
      End If
    End If
  End If
End If
Set Table = Nothing               ' Free the TTable-type object.
```

3.14.17 TTable.First Procedure

TTable.First positions a database cursor on the first record in a database table and makes it the active record.

Syntax

TTable.First

Remarks

The table must be open. Calling this method posts (i.e. writes to the table) any changes to the active record. This method is useful in combination with the TTable.Bof and TTable.Eof methods.

 **TTable**-type Objects

Example

```
Dim Table
Set Table = RDb.CreateTableObject ' Create a TTable-type object.
' Name of the database, as defined in a visualization project.
Table.DatabaseName = "Water"
If Table.OpenTable Then           ' If the current table can be opened.
```

```
  Table.First                    ' Move to the first record.
  ' While the Table.Eof method returns False.
  While Not Table.Eof
    ' ...
    Table.Next                   ' Move to the next record.
  WEnd
  Table.CloseTable               ' Close the table.
End If
Set Table = Nothing              ' Free the TTable-type object.
```

3.14.18 TTable.GetFieldValue Function

TTable.GetFieldValue returns the value of a field linked to a specified tag for the current record in a database table.

Syntax

TTable.GetFieldValue(*DevName*, *TagName*: **String): Variant**

Argument	Description
DevName	The name of the device that the tag belongs to.
TagName	The name of the tag that the field is linked to.

Return values

If the call succeeds, the method returns the value of the field for the current record.

If the call fails, the method returns **Empty**.

Remarks

The table must be open and not empty. Before calling this method, you can test whether the field exists by a call to the TTable.FieldExists method.

TTable-type Objects

Example

```
Dim Table, Count, Sum
Sum = 0
Count = 0
Set Table = RDb.CreateTableObject ' Create a TTable-type object.
```

```
' Name of the database, as defined in a visualization project.
Table.DatabaseName = "Water"
If Table.OpenTable Then           ' If the current table can be opened.
  Table.First                     ' Move to the first record.
  ' While the Table.Eof method returns False.
  While Not Table.Eof
    ' Get the value of the database field linked to the tag
    ' WaterTemperature from the device PLC1.
    Sum = Sum + Table.GetFieldValue("PLC1", "WaterTemperature")
    Count = Count + 1
    Table.Next                    ' Move to the next record.
  WEnd
  Table.CloseTable                ' Close the table.
  Set Table = Nothing             ' Free the TTable-type object.
  If Count > 0 Then               ' Check count for the value of 0.
    ' Store information on the result of the operation
    ' to the tag DisplayResult from the device System.
    RTag.SetTagValue "System", "DisplayResult", "The average value is: " + CStr(Value
/ Count)
  End If
End If
```

3.14.19 TTable.Last Procedure

TTable.Last positions a database cursor on the last record in a database table and makes it the active record.

Syntax

TTable.Last

Remarks

The table must be open. Calling this method posts (i.e. writes to the table) any changes to the active record. This method is useful in combination with the TTable.Bof and TTable.Eof methods.

TTable-type Objects

Example

```
Dim Table
Set Table = RDb.CreateTableObject ' Create a TTable-type object.
' Name of the database, as defined in a visualization project.
```

```
Table.DatabaseName = "Water"
If Table.OpenTable Then            ' If the current table can be opened.
  Table.Last                       ' Move to the last record.
  ' While the Table.Bof method returns False.
  While Not Table.Bof
    ' ...
    Table.Prior                    ' Move to the prior record.
  WEnd
  Table.CloseTable                 ' Close the table.
End If
Set Table = Nothing                ' Free the TTable-type object.
```

3.14.20 TTable.MoveBy Procedure

TTable.MoveBy positions a database cursor on a record relative to the active record in a database table and makes it the active record.

Syntax

TTable.MoveBy *Count*: **Integer**

Argument	Description
Count	The number of records by which to move the cursor. It can be either a positive (moving forward) or negative (moving backward) integer value.

Remarks

The table must be open. Calling this method posts (i.e. writes to the table) any changes to the active record. This method is useful in combination with the TTable.Bof and TTable.Eof methods.

TTable-type Objects

Example

```
Dim Table
Set Table = RDb.CreateTableObject       ' Create a TTable-type object.
' Name of the database, as defined in a visualization project.
Table.DatabaseName = "Water"
```

```
If Table.OpenTable Then                  ' If the current table can be opened.
  If Not (Table.Bof And Table.Eof) Then  ' If the table is not empty.
    Table.First                          ' Move to the first record.
    Table.MoveBy 5                       ' Move by 5 records forward.
    ' ...
  End If
  Table.CloseTable                       ' Close the table.
End If
Set Table = Nothing                      ' Free the TTable-type object.
```

3.14.21 TTable.Next Procedure

TTable.Next positions a database cursor on the next record in a database table and makes it the active record.

Syntax

TTable.Next

Remarks

The table must be open. Calling this method posts (i.e. writes to the table) any changes to the active record. This method is useful in combination with the TTable.Bof and TTable.Eof methods.

TTable-type Objects

Example

```
Dim Table
Set Table = RDb.CreateTableObject ' Create a TTable-type object.
' Name of the database, as defined in a visualization project.
Table.DatabaseName = "Water"
If Table.OpenTable Then           ' If the current table can be opened.
  Table.First                     ' Move to the first record.
  ' While the Table.Eof method returns False.
  While Not Table.Eof
    ' ...
    Table.Next                    ' Move to the next record.
  WEnd
  Table.CloseTable                ' Close the table.
End If
Set Table = Nothing               ' Free the TTable-type object.
```

3.14.22 TTable.OpenTable Function

TTable.OpenTable opens a database table.

Syntax

TTable.OpenTable: Boolean

Return values

Value	Meaning
True	The table has been opened.
False	The table has not been opened.

Remarks

Before calling this method, it is advisable to test whether the table exists by calling the TTable.TableExists method.

TTable-type Objects

Example

```
Dim Table
Set Table = RDb.CreateTableObject ' Create a TTable-type object.
' Name of the database, as defined in a visualization project.
Table.DatabaseName = "Water"
If Table.OpenTable Then           ' If the current table can be opened.
  Table.Append                    ' Append a new record.
  ' Save current system date to the new record.
  Table.DateFieldValue = Date
  ' Save current system time to the new record.
  Table.TimeFieldValue = Time
  ' Save the value of the tag WaterTemperature from the device PLC1 to the new record.
  Table.SetFieldValue "PLC1", "WaterTemperature", RTag.GetTagValue("PLC1",
"WaterTemperature")
  Table.Post                      ' Write the new record to the table.
  Table.CloseTable                ' Close the table.
End If
Set Table = Nothing               ' Free the TTable-type object.
```

3.14.23 TTable.Post Procedure

TTable.Post posts (i.e. writes to the table) any changes to a database table's active record.

Syntax

TTable.Post

Remarks

The table must be open and must be in edit or insert mode. To put a table into edit or insert mode, call the TTable.Edit or TTable.Append method.

TTable-type Objects

Example

```
Dim Table
Set Table = RDb.CreateTableObject ' Create a TTable-type object.
' Name of the database, as defined in a visualization project.
Table.DatabaseName = "Water"
If Table.OpenTable Then           ' If the current table can be opened.
  Table.Append                    ' Append a new record.
  ' Save current system date to the new record.
  Table.DateFieldValue = Date
  ' Save current system time to the new record.
  Table.TimeFieldValue = Time
  ' Save the value of the tag WaterTemperature from the device PLC1 to the new record.
  Table.SetFieldValue "PLC1", "WaterTemperature", RTag.GetTagValue("PLC1",
"WaterTemperature")
  Table.Post                      ' Write the new record to the table.
  Table.CloseTable                ' Close the table.
End If
Set Table = Nothing               ' Free the TTable-type object.
```

3.14.24 TTable.Prior Procedure

TTable.Prior positions a database cursor on the previous record in a database table and makes it the active record.

Syntax

TTable.Prior

Remarks

The table must be open. Calling this method posts (i.e. writes to the table) any changes to the active record. This method is useful in combination with the TTable.Bof and TTable.Eof methods.

TTable-type Objects

Example

```
Dim Table
Set Table = RDb.CreateTableObject ' Create a TTable-type object.
' Name of the database, as defined in a visualization project.
Table.DatabaseName = "Water"
If Table.OpenTable Then           ' If the current table can be opened.
  Table.Last                      ' Move to the last record.
  ' While the Table.Bof method returns False.
  While Not Table.Bof
    ' ...
    Table.Prior                   ' Move to the prior record.
  WEnd
  Table.CloseTable                ' Close the table.
End If
Set Table = Nothing               ' Free the TTable-type object.
```

3.14.25 TTable.SetFieldValue Procedure

TTable.SetFieldValue sets the value of a field linked to a specified tag to a specified value for the current record in a database table.

Syntax

TTable.SetFieldValue *DevName*, *TagName*: **String**; *Value*: **Variant**

Argument	Description
DevName	The name of the device that the tag belongs to.
TagName	The name of the tag that the field is linked to.
Value	The new value for the field.

Remarks

The table must be open and must be in edit or insert mode. To put a table into edit or insert mode, call the TTable.Edit or TTable.Append method. Before calling this method, it is advisable to test whether the field exists by calling the TTable.FieldExists method. After modifying the active record using this method, call the TTable.Post method to post the record (i.e. write it to the table). Setting the active record to another record using the TTable.First, TTable.Last, TTable.Next, TTable.Prior, TTable.MoveBy methods also posts the record. If the record has not yet been posted, the changes to the record can be canceled by calling the TTable.Cancel method or closing the table using the TTable.CloseTable method.

TTable-type Objects

Example

```
Dim Table
Set Table = RDb.CreateTableObject ' Create a TTable-type object.
' Name of the database, as defined in a visualization project.
Table.DatabaseName = "Water"
If Table.OpenTable Then           ' If the current table can be opened.
  Table.Append                    ' Append a new record.
  ' Save current system date to the new record.
  Table.DateFieldValue = Date
  ' Save current system time to the new record.
```

```
  Table.TimeFieldValue = Time
  ' Save the value of the tag WaterTemperature from the device PLC1 to the new record.
  Table.SetFieldValue "PLC1", "WaterTemperature", RTag.GetTagValue("PLC1",
"WaterTemperature")
  Table.Post                        ' Write the new record to the table.
  Table.CloseTable                  ' Close the table.
End If
Set Table = Nothing                 ' Free the TTable-type object.
```

3.14.26 TTable.TableExists Function

TTable.TableExists indicates whether a database table exists.

Syntax

TTable.TableExists: Boolean

Return values

Value	Meaning
True	The table exists.
False	The table does not exist.

Remarks

If the table does not exist, it can be created by a call to the TTable.CreateTable method.

TTable-type Objects

Example

```
Dim Table
Set Table = RDb.CreateTableObject ' Create a TTable-type object.
' Name of the database, as defined in a visualization project.
Table.DatabaseName = "Database1"
' We want to access an archive table that contains data from August, 2000.
Table.IsArchive = True
Table.ArchiveName = "C:\Reliance\Projects\Test\Data\2000\d1_0008.DB"
If Table.TableExists Then           ' If the archive table exists.
  If Table.OpenTable Then           ' If the table can be opened.
    ' ...
```

```
    Table.CloseTable              ' Close the table.
  End If
End If
Set Table = Nothing               ' Free the TTable-type object.
```

3.14.27 TTable.UpdateTableStructure Procedure

TTable.UpdateTableStructure updates the structure of a database table by adding new columns to the table based on data table fields defined via the *Data Table Manager* (if any fields were added later).

Syntax

TTable.UpdateTableStructure

Remarks

The table must exist. It does not matter if it is open or closed. After calling the procedure, the table is closed.

 **TTable**-type Objects

Example

```
Dim Table
Set Table = RDb.CreateTableObject ' Create a TTable-type object.
' Name of the database, as defined in a visualization project.
Table.DatabaseName = "Water"
If Table.TableExists Then         ' If the table exists.
  Table.UpdateTableStructure      ' Update the table's structure.
  If Table.OpenTable Then         ' If the table can be opened.
    ' ...
    Table.CloseTable              ' Close the table.
  End If
End If
Set Table = Nothing               ' Free the TTable-type object.
```

3.15 RTag Object

The **RTag** object implements methods for operations on tags defined in a visualization project. The tags can be of simple data types (e.g. *Word*, *Byte*, *Bool*, *String*, etc.) or array data types (e.g. *Array of Word*).

Methods:

- **RTag.SetTagElementValues** Procedure
- **RTag.GetTagElementValue** Function
- **RTag.GetTagValue** Function
- **RTag.MoveTagElementValues** Procedure
- **RTag.MoveTagElementValuesToSimpleTag** Procedure
- **RTag.MoveTagValue** Procedure
- **RTag.MoveTagValueToArrayTag** Procedure
- **RTag.SetTagElementValue** Procedure
- **RTag.SetTagValue** Procedure
- **RTag.UpdateTagValue** Procedure

3.15.1 RTag.SetTagElementValues Procedure

RTag.SetTagElementValues fills the value of an array-type tag or its part with a specified value.

Syntax

RTag.SetTagElementValues *DevName*, *TagName*: **String; Value: Variant;** *Offset*, *Count*: **Integer**

Argument	Description
DevName	The name of the device that the tag belongs to.
TagName	The name of the tag.
Value	The value to be used for the array elements.

Offset	The zero-based index of the first array element to be set to the specified value.
Count	The number of array elements to be set to the specified value.

Remarks

The tag must be of an array data type (e.g. *Array of Word*). The value of the *Offset* and *Count* arguments is checked at runtime, so calling the method never affects memory not belonging to the tag.

RTag Object

Example

```
' Fill 20 elements of the tag ByteArray from the device PLC1
' with the value 0 starting with the element at the index 10.
RTag.SetTagElementValues "PLC1", "ByteArray", 0, 10, 20
```

3.15.2 RTag.GetTagElementValue Function

RTag.GetTagElementValue returns the value of an array-type tag at a specified index.

Syntax

RTag.GetTagElementValue(*DevName***, *TagName*: String; *Index*: Integer): Variant**

Argument	Description
DevName	The name of the device that the tag belongs to.
TagName	The name of the tag.
Index	The zero-based index of the array element whose value is to be returned.

Return values

If the call succeeds, the method returns the value of the tag at the specified index.

If the call fails, the method returns **Empty**.

Remarks

The tag must be of an array data type (e.g. *Array of Word*). To find out whether the return value is valid, use the IsEmpty function. If IsEmpty returns **True**, the return value of **RTag.GetTagElementValue** has not been assigned a value (e.g. if the specified tag does not exist, is of an incorrect data type, has an invalid value, etc.). The validity of the return value of **RTag.GetTagElementValue** is also affected by the *Return value even if not valid* option (*Reliance Design* > *Project* > *Options* > *Scripts* > *Other*).

 **RTag** Object

Example

```
Dim Value
' Get the value of the tag WordArray
' from the device PLC1 at the index 0.
Value = RTag.GetTagElementValue("PLC1", "WordArray", 0)
If Not IsEmpty(Value) Then ' Check for the validity of the result.
  ' ...
End If
```

3.15.3 RTag.GetTagValue Function

RTag.GetTagValue returns the value of a tag.

Syntax

RTag.GetTagValue(*DevName***,** *TagName***: String): Variant**

Argument	Description
DevName	The name of the device that the tag belongs to.
TagName	The name of the tag.

Return values

If the call succeeds, the method returns the value of the tag.

If the call fails, the method returns **Empty**.

Remarks

The tag must be of a simple data type (e.g. *Bool*, *Byte*, *Word*, *String*, etc.). To find out whether the return value is valid, use the IsEmpty function. If IsEmpty returns **True**, the return value of **RTag.GetTagValue** has not been assigned a value (e.g. if the specified tag does not exist, is of an incorrect data type, has an invalid value, etc.). The validity of the return value of **RTag.GetTagValue** is also affected by the *Return value even if not valid* option (*Reliance Design* > *Project* > *Options* > *Scripts* > *Other*).

RTag Object

Example

```
Dim Value
' Get the value of the tag Var1 from the device PLC1.
Value = RTag.GetTagValue("PLC1", "Var1")
If Not IsEmpty(Value) Then ' Check for the validity of the result.
  ' ...
End If
```

3.15.4 RTag.MoveTagElementValues Procedure

RTag.MoveTagElementValues copies the value of an array-type tag or its part to another array-type tag.

Syntax

RTag.MoveTagElementValues *SourceDevName*, *SourceTagName*, *TargetDevName*, *TargetTagName*: **String;** *SourceIndex*, *TargetIndex*, *Count*: **Integer**

Argument	Description
SourceDevName	The name of the device that the source tag belongs to.
SourceTagName	The name of the source tag.
TargetDevName	The name of the device that the target tag belongs to.
TargetTagName	The name of the target tag.

SourceIndex	The zero-based index of the first element of the source array to be copied.
TargetIndex	The zero-based index of the first element of the target array to be overwritten.
Count	The number of array elements to be copied.

Remarks

The tags must be of an array data type. The base data type (i.e. data type of array elements) of the source and target tag may differ, but must have the same size (e.g. *Array of DoubleWord* and *Array of LongInt*). However, if the source tag is of type *Array of Bool*, the target tag must be of the same type. When copying the value of individual array elements, no data type conversion is performed. Also, the *Analog value correction* and *Negate value if digital* properties are ignored (*Reliance Design* > *Managers* > *Device Manager* > tag properties > the *Correction* page). Thus, after calling **RTag.MoveTagElementValues**, the values of the corresponding elements of the source and target array can differ because of different interpretation.

 **RTag** Object

Example

```
' Copy 10 elements of the tag WordArray from the device System
' starting with the element at the index 0
' to the tag WordArray from the device Virtual1
' starting with the element at the index 50.
RTag.MoveTagElementValues "System", "WordArray", "Virtual1", "WordArray", 0, 50, 10
```

3.15.5 RTag.MoveTagElementValuesToSimpleTag Procedure

RTag.MoveTagElementValuesToSimpleTag copies the value of an array-type tag or its part to a simple-type tag (e.g. *Bool*, *Byte*, *Word*, *String*, etc.).

Syntax

RTag.MoveTagElementValuesToSimpleTag *SourceDevName*, *SourceTagName*, *TargetDevName*, *TargetTagName*: **String;** *SourceIndex*: **Integer**

Argument	Description
SourceDevName	The name of the device that the source tag belongs to.
SourceTagName	The name of the source tag.
TargetDevName	The name of the device that the target tag belongs to.
TargetTagName	The name of the target tag.
SourceIndex	The zero-based index of the first element of the source array to be copied.

Remarks

The source and target tag data types must meet the following requirements:

- the target tag data type size must be dividable by the size of the base data type (i.e. data type of array elements) of the source tag

- if the source tag is of type *Array of Bool*, the target tag must be of type *Bool*

When copying the value, no data type conversion is performed. Also, the *Analog value correction* and *Negate value if digital* properties are ignored (*Reliance Design* > *Managers* > *Device Manager* > tag properties > the *Correction* page). The number of array elements copied depends on the target tag data type size.

RTag Object

Example

```
' Copy 4 elements of the tag ByteArray from the device System
' starting with the element at the index 0 to the tag Float.
' The number of elements copied is determined by the size of the tag Float.
RTag.MoveTagElementValuesToSimpleTag "System", "ByteArray", "System", "Float", 0
```

3.15.6 RTag.MoveTagValue Procedure

RTag.MoveTagValue copies the value of a tag to another tag.

Syntax

RTag.MoveTagValue *SourceDevName*, *SourceTagName*, *TargetDevName*, *TargetTagName*: `String`

Argument	Description
SourceDevName	The name of the device that the source tag belongs to.
SourceTagName	The name of the source tag.
TargetDevName	The name of the device that the target tag belongs to.
TargetTagName	The name of the target tag.

Remarks

The tags must be of a simple data type (e.g. *Bool*, *Byte*, *Word*, *String*, etc.). The data type of the source and target tag may differ, but must have the same size (e.g. *DoubleWord* and *LongInt*). However, if the source tag is of type *Bool*, the target tag must be of the same type. When copying the value, no data type conversion is performed. Also, the *Analog value correction* and *Negate value if digital* properties are ignored (*Reliance Design* > *Managers* > *Device Manager* > tag properties > the *Correction* page). Thus, after calling **RTag.MoveTagValue**, the value of the source and target tag can differ because of different interpretation.

RTag Object

Example

```
' Copy the value of the tag Word from the device System
' to the tag Word from the device Virtual1.
RTag.MoveTagValue "System", "Word", "Virtual1", "Word"
```

3.15.7 RTag.MoveTagValueToArrayTag Procedure

RTag.MoveTagValueToArrayTag copies the value of a simple-type tag (e.g. *Bool*, *Byte*, *Word*, *String*, etc.) to an array-type tag.

Syntax

RTag.MoveTagValueToArrayTag *SourceDevName*, *SourceTagName*, *TargetDevName*, *TargetTagName*: **String**; *TargetIndex*: **Integer**

Argument	Description
SourceDevName	The name of the device that the source tag belongs to.
SourceTagName	The name of the source tag.
TargetDevName	The name of the device that the target tag belongs to.
TargetTagName	The name of the target tag.
TargetIndex	The zero-based index of the first element of the target array to be overwritten.

Remarks

The source and target tag data types must meet the following requirements:

- the source tag data type size must be dividable by the size of the base data type (i.e. data type of array elements) of the target tag

- if the source tag is of type *Bool*, the target tag must be of type *Array of Bool*

When copying the value, no data type conversion is performed. Also, the *Analog value correction* and *Negate value if digital* properties are ignored (*Reliance Design* > *Managers* > *Device Manager* > tag properties > the *Correction* page). The number of array elements overwritten depends on the source tag data type size.

RTag Object

Example

```
' Copy the value of the tag Float from the device System
```

```
' to the tag ByteArray from the device System
' starting with the element at the index 0.
' The number of elements overwritten is determined by the size of the tag Float.
RTag.MoveTagValueToArrayTag "System", "Float", "System", "ByteArray", 0
```

3.15.8 RTag.SetTagElementValue Procedure

RTag.SetTagElementValue sets the value of an array-type tag at a specified index to a specified value.

Syntax

RTag.SetTagElementValue *DevName*, *TagName*: **String;** *Index*: **Integer;** *Value*: **Variant**

Argument	Description
DevName	The name of the device that the tag belongs to.
TagName	The name of the tag.
Index	The zero-based index of the array element whose value is to be set.
Value	The new value for the array element.

Remarks

The tag must be of an array data type (e.g. *Array of Word*). The form of specifying the value depends on the tag's data type:

Data type	Form
Integer-type arrays	A sequence of digits.
Floating point-type arrays	A sequence of digits which may include the period character as a decimal separator.
String-type arrays	A sequence of characters enclosed in double quotes.
Digital-type arrays	One of the constants **False**, **True** (0, 1).

RTag Object

Example

```
' Set the value of the tag WordArray (an integer-type array)
' from the device PLC1 at the index 0 to the value 20.
RTag.SetTagElementValue "PLC1", "WordArray", 0, 20
' Set the value of the tag FloatArray (a floating point-type array)
' from the device PLC1 at the index 0 to the value 100.25.
RTag.SetTagElementValue "PLC1", "FloatArray", 0, 100.25
' Set the value of the tag StringArray (a string-type array)
' from the device PLC1 at the index 0 to the value "Hello".
RTag.SetTagElementValue "PLC1", "StringArray", 0, "Hello"
' Set the value of the tag BoolArray (a digital-type array)
' from the device PLC1 at the index 10 to the value True.
RTag.SetTagElementValue "PLC1", "BoolArray", 10, True
```

3.15.9 RTag.SetTagValue Procedure

RTag.SetTagValue sets the value of a tag to a specified value.

Syntax

RTag.SetTagValue *DevName*, *TagName*: **String**; *Value*: **Variant**

Argument	Description
DevName	The name of the device that the tag belongs to.
TagName	The name of the tag.
Value	The new value for the tag.

Remarks

The tag must be of a simple data type (e.g. *Bool*, *Byte*, *Word*, *String*, etc.). The form of specifying the value depends on the tag's data type:

Data type	Form
Integer types	A sequence of digits.

Floating point types	A sequence of digits which may include the period character as a decimal separator.
String type	A sequence of characters enclosed in double quotes.
Digital type	One of the constants **False**, **True** (0, 1).

RTag Object

Example

```
' Set the value of the tag Word (an integer-type tag)
' from the device PLC1 to the value 20.
RTag.SetTagValue "PLC1", "Word", 20
' Set the value of the tag Float (a floating point-type tag)
' from the device PLC1 to the value 100.25.
RTag.SetTagValue "PLC1", "Float", 100.25
' Set the value of the tag String (a string-type tag)
' from the device PLC1 to the value "Hello".
RTag.SetTagValue "PLC1", "String", "Hello"
' Set the value of the tag Bool (a digital-type tag)
' from the device PLC1 to the value 0.
RTag.SetTagValue "PLC1", "Bool", 0
```

3.15.10 RTag.UpdateTagValue Procedure

RTag.UpdateTagValue updates the value of a tag.

Syntax

RTag.UpdateTagValue *DevName*, *TagName*: **String**

Argument	Description
DevName	The name of the device that the tag belongs to.
TagName	The name of the tag.

Remarks

The way in which the value is updated depends on the type of the device the tag belongs to and the way in which the device is connected to the computer:

A physical device directly connected to the computer (e.g. PLC connected via a serial cable)	Calling the method forces the communication driver to reread the value from the device (this feature might not be supported by every communication driver).
A virtual device directly connected to the computer or the System device	Tags belonging to such a device are always up-to-date. Calling the method has no effect.
A device provided to the computer through a network connection	Calling the method forces the runtime software to reread the value from the server computer.

 **RTag** Object

Example

```
' Update the value of the tag Word from the device PLC1.
RTag.UpdateTagValue "PLC1", "Word"
```

3.16 RUser Object

The **RUser** object implements methods for operations on users defined in a visualization project. The methods enable you to log users on and off of the runtime software, retrieve the name of the currently logged on user, and retrieve information on users. The methods only operate on users connected to the computer on which a visualization project is running.

Methods:

- **RUser.CheckUserAccessRights** Function
- **RUser.CheckUserPassword** Function
- **RUser.GetLoggedOnUserName** Function
- **RUser.GetUserID** Function
- **RUser.IsUserAdmin** Function
- **RUser.LogOffUser** Procedure
- **RUser.LogOnUser** Procedure
- **RUser.LogOnUserWithCode** Function
- **RUser.LogOnUserWithNameAndPassword** Function
- **RUser.UserExists** Function

3.16.1 RUser.CheckUserAccessRights Function

RUser.CheckUserAccessRights determines whether a user has at least one of specified access rights.

Syntax

RUser.CheckUserAccessRights(*User***: Variant;** *Rights***: String): Boolean**

Argument	Description
User	The name or ID of the user.
Right	The names of the access rights of interest separated with commas.

Remarks

The method only operates on users connected to the computer on which a visualization project is running.

Return values

Value	Meaning
True	The user has at least one of the specified access rights.
False	The user does not exist or does not have any of the specified access rights.

 **RUser** Object

Example

```
Dim UserName
' If a user is logged on.
If RUser.GetLoggedOnUserName(UserName) Then
  ' If the user has the Servicing (denoted by "S") or "Terminate" access right.
  If RUser.CheckUserAccessRights(UserName, "S,Terminate") Then
    ' ...
  End If
End If
```

3.16.2 RUser.CheckUserPassword

RUser.CheckUserPassword determines whether a user has a specified password.

Syntax

RUser.CheckUserPassword(*User***: Variant;** *Password***: String): Boolean**

Argument	Description
User	The name or ID of the user.
Password	User password.

Remarks

The method only operates on users connected to the computer on which a visualization project is running.

Return values

Value	Meaning
True	The user has specified password.
False	The user does not exist or does not have specified password.

RUser Object

Example

```
Dim UserName
' If a user is logged on.
If RUser.GetLoggedOnUserName(UserName) Then
  ' If the user has "123456" password.
  If RUser.CheckUserPassword(UserName, "123456") Then
    ' ...
  End If
End If
```

3.16.3 RUser.GetLoggedOnUserName Function

RUser.GetLoggedOnUserName returns the name of the user currently logged on to the runtime software.

Syntax

RUser.GetLoggedOnUserName(ByRef *UserName*: **Variant): Boolean**

Argument	Description
UserName	A variable that is to receive the name of the user.

Return values

Value	Meaning
True	A user is logged on.
False	No user is logged on.

 **RUser** Object

Example

```
Dim UserName
' If a user is logged on.
If RUser.GetLoggedOnUserName(UserName) Then
  ' If the name of the user is "Service".
  If UserName = "Service" Then
    ' ...
  End If
End If
```

3.16.4 RUser.IsUserAdmin Function

RUser.IsUserAdmin determines whether a user is a user administrator (i.e. whether the user is allowed to administrate users; *User Manager* > user properties > the *Basic* page).

Syntax

RUser.IsUserAdmin(*User***: Variant): Boolean**

Argument	Description
User	The name or ID of the user.

Return values

Value	Meaning
True	The user is a user administrator.

False	The user does not exist or is not a user administrator.

Remarks

The method only operates on users connected to the computer on which a visualization project is running.

RUser Object

Example

```
Dim UserName
' If a user is logged on.
If RUser.GetLoggedOnUserName(UserName) Then
  ' If the user is a user administrator.
  If RUser.IsUserAdmin(UserName) Then
    ' ...
  End If
End If
```

3.16.5 RUser.GetUserID Function

RUser.GetUserID returns the ID (a unique integer identifier) for a user.

Syntax

RUser.GetUserID(*UserName***: String): Integer**

Argument	Description
UserName	The name of the user.

Return values

If the call succeeds, the method returns the ID of the user.

If the call fails, the method returns 0.

Remarks

The method only operates on users connected to the computer on which a visualization project is running.

RUser Object

Example

```
Dim UserID
If RUser.UserExists("Operator") Then   ' If a user of the name Operator exists.
  UserID = RUser.GetUserID("Operator") ' Get the ID of the user.
  ' ...
End If
```

3.16.6 RUser.LogOffUser Procedure

RUser.LogOffUser logs off the user currently logged on to the runtime software.

Syntax

RUser.LogOffUser

RUser Object

Example

```
Dim UserName
' If the tag EndOfShift from the device System is equal to True.
If RTag.GetTagValue("System", "EndOfShift") Then
  RUser.LogOffUser ' Log off the user.
End If
```

3.16.7 RUser.LogOnUser Procedure

RUser.LogOnUser displays a log-on user dialog to enable the user to log on to the runtime software.

Syntax

RUser.LogOnUser

RUser Object

Example

```
Dim UserName
If Not RUser.GetLoggedOnUserName(UserName) Then
  ' If no user is currently logged on, display a log-on user dialog.
  RUser.LogOnUser
End If
```

3.16.8 RUser.LogOnUserWithCode Function

RUser.LogOnUserWithCode logs on the user whose *Code* property matches a specified code (*User Manager* > user properties > the *Basic* page).

Syntax

RUser.LogOnUserWithCode(*Code***: String): Boolean**

Argument	Description
Code	The code of the user to log on.

Remarks

The method only operates on users connected to the computer on which a visualization project is running.

Return values

Value	Meaning
True	A user with the specified code exists.
False	A user with the specified code does not exist.

RUser Object

Example

```
Dim UserCode
' Get the value of the tag UserCode from the device System.
UserCode = RTag.GetTagValue("System", "UserCode")
If RUser.LogOnUserWithCode(UserCode) Then
  ' ...
End If
```

3.16.9 RUser.LogOnUserWithNameAndPassword Function

RUser.LogOnUserWithNameAndPassword logs on the user whose *Name* and *Password* properties match specified credentials (*User Manager* > user properties > the *Basic* page).

Syntax

RUser.LogOnUserWithNameAndPassword(*UserName***, *Password*: String): Boolean**

Argument	Description
UserName	The name of the user to log on.
Password	The password of the user to log on.

Remarks

The method only operates on users connected to the computer on which a visualization project is running.

Return values

Value	Meaning
True	A user with the specified credentials exists.
False	A user with the specified credentials does not exist.

RUser Object

Example

```
Dim UserName, Password
' Get the value of the tag UserName from the device System.
UserName = RTag.GetTagValue("System", "UserName")
' Get the value of the tag Password from the device System.
Password = RTag.GetTagValue("System", "Password")
If RUser.LogOnUserWithNameAndPassword(UserName, Password) Then
  ' ...
End If
```

3.16.10 RUser.UserExists Function

RUser.UserExists determines whether a specified user exists.

Syntax

RUser.UserExists(*User***: Variant): Boolean**

Argument	Description
User	The name or ID of the user.

Return values

Value	Meaning
True	The user exists.
False	The user does not exist.

Remarks

The method only considers users connected to the computer on which a visualization project is running.

RUser Object

Example

```
Dim UserID
If RUser.UserExists("Operator") Then   ' If a user of the name Operator exists.
  UserID = RUser.GetUserID("Operator") ' Get the ID of the user.
  ' ...
End If
```

3.17 RWS Object

The **RWS** object implements methods for accessing the Web service of Reliance data servers.

Methods:

RWS.GetThinClientList Procedure

3.17.1 RWS.GetThinClientList Procedure

RWS.GetThinClientList returns a list of objects containing information about thin clients connected to the data server.

Syntaxe

```
RWS.GetThinClientList ByRef ClientList: Variant, ByRef ClientCount: Variant
```

Argument	Popis
ClientList	List of connected thin clients.
ClientCount	Number of connected thin clients.

RWS Object

Example

```
Dim ClientList, ClientCount, ClientIndex, ClientInfo
' Retrieving the list of thin clients
RWS.GetThinClientList ClientList, ClientCount
RTag.SetTagValue "System", "ThinClients_Count", ClientCount
For ClientIndex = 0 To ClientCount - 1
  Set ClientInfo = ClientList(ClientIndex)
  RTag.SetTagElementValue "System", "ThinClients_SessionId", ClientIndex, ClientInfo.SessionId
  RTag.SetTagElementValue "System", "ThinClients_IPAddress", ClientIndex, ClientInfo.IPAddress
  RTag.SetTagElementValue "System", "ThinClients_SoftwareType", ClientIndex, ClientInfo.SoftwareType ' Web Client = 0, Mobile Client = 1
  RTag.SetTagElementValue "System", "ThinClients_SoftwareVersion", ClientIndex, ClientInfo.SoftwareVersion
  RTag.SetTagElementValue "System", "ThinClients_ComputerId", ClientIndex, ClientInfo.
```

```
ComputerId
  RTag.SetTagElementValue "System", "ThinClients_ComputerName", ClientIndex,
ClientInfo.ComputerName
  RTag.SetTagElementValue "System", "ThinClients_UserId", ClientIndex, ClientInfo.
UserId
  RTag.SetTagElementValue "System", "ThinClients_UserName", ClientIndex, ClientInfo.
UserName
  RTag.SetTagElementValue "System", "ThinClients_RegisterDateTime", ClientIndex,
ClientInfo.RegisterDateTime
  RTag.SetTagElementValue "System", "ThinClients_LastRequestDateTime", ClientIndex,
ClientInfo.LastRequestDateTime
  RTag.SetTagElementValue "System", "ThinClients_RequestCount", ClientIndex,
ClientInfo.RequestCount
  RTag.SetTagElementValue "System", "ThinClients_Disconnected", ClientIndex,
ClientInfo.Disconnected
  Set ClientInfo = Nothing
Next
```