Calculated Fields Scripting Guide

Revised: December 6th, 2022

Techno Solutions

Table of Contents

Overview

Calculated Fields are the fields that derive their values using specified calculation performed on values of other fields of a record.

Calculated Fields can be of two types:

- **Text fields:** This type of field can display the result of a calculation formula that gives a text value. For example, you can concatenate values of two fields.

- **Numeric (Decimal fields):** This type of field can display the result of a calculation formula that outputs a numeric value. For example, you can perform mathematical operations on numeric fields to calculate the value of a numeric calculated field.

How to define Calculated Fields?

After learning the concepts and technical features of *Calculated Fields* from this document, you can learn about creating and using *Calculated Fields* in the TopTeam repository using this help article: Configuring and using Calculated Fields.

Programming Languages for Scripting

For scripting *Calculated Fields* you use one of the two supported programming languages:

- **Basic**
- **Pascal**

Basic Syntax

Overview

· **sub .. end** and **function .. end** declarations

· **byref** and **dim** directives

· **if .. then .. else .. end** constructor

· **for .. to .. step .. next** constructor

· **do .. while .. loop** and **do .. loop .. while** constructors

· **do .. until .. loop** and **do .. loop .. until** constructors

· **^ , * , / , and , + , - , or , <> , >=, <= , = , > , < , div , mod , xor , shl , shr** operators

· **try .. except** and **try .. finally** blocks

· **select case .. end select** constructor

· **array** constructors (x:=[1, 2, 3];)

· **exit** statement

· access to object properties and methods (**ObjectName.SubObject.Property**)

Script structure

One line Script

If you just have a one line calculation formula to be processed, you can write a one line script to set value in the implicitly declared variable "Output".

Examples:

Script1:
```
Output = GetFieldValue("Name") + GetFieldValue("State")
```

Script2:
```
Output = GetWeightageOf("Priority")* GetWeightageOf("Complexity")
```

Multiline Script

Here the structure is made of two major blocks:

a) Main block

b) Procedure and function declarations

Main block is mandatory. There is no requirement for the main block to be put inside a *begin..end* pair. It can be a single statement. The script should set the result in a special variable called "Output".

Examples:

SCRIPT 1:

```
' This is a function declaration
FUNCTION GetNameAndState

  GetNameAndState = GetFieldValue("Name") + GetFieldValue("State")
END FUNCTION

Output = GetNameAndState //This is main block
```

SCRIPT 2:

```
' This is a method declaration
SUB SetNameAndState
  Output = GetFieldValue('Name') + GetFieldValue('State');
END SUB

' This is main block
SetNameAndState
```

Like in normal Basic script, statements in a single line can be separated by ":" character.

NOTE	The script must set the result in the implicitly declared variable "Output" to display it in the calculated field.

Identifiers

Identifier names in script (variable names, function and procedure names, etc.) follow the most common rules in Basic: should begin with a character (a..z or A..Z), or '_', and can be followed by alphanumeric chars or '_' char. Cannot contain any other character or spaces.

Valid identifiers:

```
VarName
_Some
V1A2

_____Some____
```

Invalid identifiers:

```
2Var
My Name

Some-more
This,is,not,valid
```

Assign statements

Assign statements (assign a value or expression result to a variable or object property) are built using "=".
Example:

```
MyVar = 2
Button.Caption = "This " + "is ok."
```

Character strings

Strings (sequence of characters) are declared in Basic using double quote (") character.
Examples:

```
A = "This is a text"
Str = "Text "+"concat"
```

Comments

Comments can be inserted inside script. You can use ' chars or REM. Comment will finish at the end of line.

Examples:

```
' This is a comment before ShowMessage
ShowMessage("Ok")

REM This is another comment
ShowMessage("More ok!")

' And this is a comment
' with two lines
ShowMessage("End of okays")
```

Variables

There is no need to declare variable types in script. Thus, you declare variable just using DIM directive and its name. There is no need to declare variables if scripter property OptionExplicit is set to false. In this case, variables are implicit declared. If you want to have more control over the script, set OptionExplicit property to true.
This will raise a compile error if variable is used but not declared in script.

Examples:

SCRIPT 1:

```
SUB Msg
   DIM S
   S = "Hello world!"
   ShowMessage(S)
END SUB
```

SCRIPT 2:

```
DIM A
A = 0
A = A+1
ShowMessage(A)
```

NOTE	If script property OptionExplicit is set to false, then variable declarations are not necessary in any of the scripts above.

Indexes

Strings, arrays and array properties can be indexed using "[" and "]" chars. For example, if Str is a string variable, the expression Str[3] returns the third character in the string denoted by Str, while Str[I + 1] returns the character immediately after the one indexed by I.

Examples:

```
MyChar = MyStr[2]
MyStr[1] = "A"
MyArray[1,2] = 1530
Lines.Strings[2] = "Some text"
```

Arrays

Script support array constructors and support to variant arrays. To construct an array, use "[" and "]" chars. You can construct multi-index array nesting array constructors. You can then access arrays using indexes. If array is multi-index, separate indexes using ",".

If variable is a variant array, script automatically support indexing in that variable. A variable is a variant array is it was assigned using an array constructor, if it is a direct reference to a Delphi variable which is a variant array (see Delphi integration later) or if it was created using VarArrayCreate procedure.

Arrays in script are 0-based index.

Examples:

```
NewArray = [ 2,4,6,8 ]
Num = NewArray[1] //Num receives "4"
MultiArray = [ ["green","red","blue"] ,
["apple","orange","lemon"] ]
Str = MultiArray[0,2] //Str receives 'blue'
```

```
MultiArray[1,1] = "new orange"
```

If statements

There are two forms of if statement: if...then..end if and the if...then...else..end if. Like normal Basic, if the if expression is true, the statements are executed. If there is else part and expression is false, statements after else are executed.

Examples:

```
IF J <> 0 THEN Result = I/J END IF
IF J = 0 THEN Exit ELSE Result := I/J END IF
IF J <> 0 THEN
   Result = I/J
   Count = Count + 1
ELSE
   Done = True
END IF
```

While statements

A While statement is used to repeat statements, while a control condition (expression) is evaluated as true. The control condition is evaluated before the statements. Hence, if the control condition is false at first iteration, the statement sequence is never executed. The while statement executes its constituent statement repeatedly, testing expression before each iteration. As long as expression returns true, execution continues.

Examples:

```
WHILE (Data[I] <> X) I = I + 1 END WHILE
WHILE (I > 0)
   IF Odd(I) THEN Z = Z * X END IF
   X = Sqr(X)
END WHILE

WHILE (not Eof(InputFile))
   Readln(InputFile, Line)
   Process(Line)
END WHILE
```

Loop statements

Scripter support loop statements. The possible syntax are:

```
DO WHILE expr statements LOOP
DO UNTIL expr statements LOOP
DO statements LOOP WHILE expr
DO statement LOOP UNTIL expr
```

Statements will be execute WHILE expr is true, or UNTIL expr is true. If expr is before statements, the control condition will be tested before iteration. Otherwise, control condition will be tested after iteration.

Examples:

```
DO
   K = I mod J
   I  = J
   J  = K
LOOP UNTIL J = 0

DO UNTIL I >= 0
   Write("Enter a value (0..9): ")
   Readln(I)
LOOP

DO
K     = I mod J
I     = J
J     = K
LOOP WHILE J <> 0

DO WHILE I < 0
   Write("Enter a value (0..9): ")
   Readln(I)
LOOP
```

For statements

Scripter support for statements with the following syntax:

FOR counter = initialValue TO finalValue

STEP stepValue statements NEXT. For statement set counter to initialValue, repeats execution of statement until "next" and increment value of counter by stepValue, until counter reaches finalValue.
Step part is optional, and if omitted stepValue is considered 1.

Examples:

SCRIPT 1:

```
FOR c = 1 TO 10 STEP 2
      a = a + c
NEXT
```

SCRIPT 2:

```
FOR I = a TO b
      j = i ^ 2    sum = sum + j
NEXT
```

Select case statements

Scripter support select case statements with following syntax:

```
SELECT CASE selectorExpression
 CASE caseexpr1
   statement1
 …
 CASE caseexprn
   statementn
 CASE ELSE
   elsestatement
END SELECT
```

If selectorExpression matches the result of one of caseexprn expressions, the respective statements will be execute. Otherwise, elsestatement will be executed. Else part of case statement is optional.
Example:

```
SELECT CASE uppercase(Fruit)
   CASE "lime" ShowMessage("green")
   CASE "orange"
      ShowMessage("orange")
   CASE "apple" ShowMessage("red")
CASE ELSE
   ShowMessage("black")
END SELECT
```

Function and sub declaration

Declaration of functions and subs are similar to Basic. In functions to return function values, use implicit declared variable which has the same name of the function. Parameters by reference can also be used, using BYREF directive.

Examples:

```
SUB HelloWord
   ShowMessage("Hello world!")
END SUB

SUB UpcaseMessage(Msg)
   ShowMessage(Uppercase(Msg))
END SUB

FUNCTION TodayAsString
   TodayAsString = DateToStr(Date)
END FUNCTION

FUNCTION Max(A,B)
   IF A>B THEN
      MAX = A
   ELSE
      MAX = B
   END IF
END FUNCTION

SUB SwapValues(BYREF A, B)
   DIM TEMP
   TEMP = A
A     = B
B     = TEMP
END SUB
```

Pascal Syntax

Overview

Current Pascal syntax supports:

· **begin .. end** constructor

· **procedure** and **function** declarations

· **if .. then .. else** constructor

· **for .. to .. do .. step** constructor

· **while .. do** constructor

· **repeat .. until** constructor

· **try .. except** and **try .. finally** blocks

· **case** statements

· **array** constructors (x:=[1, 2, 3];)

· **^ , * , / , and , + , - , or , <> , >=, <= , = , > , < , div , mod , xor , shl , shr** operators

· access to object properties and methods (**ObjectName.SubObject.Property**)

Script structure

One line Script

If you just have a one line calculation formula to be processed, you can write a one line script to set value in implicitly declared variable "Output".

Example:

Script1:

```
Output := GetFieldValue('Name') + GetFieldValue('State');
```

Script2:

```
Output := GetWeightageOf('Priority')*  GetWeightageOf('Complexity');
```

Multiline Script

Here the structure is made of two major blocks: a) main block and b) procedure and function declarations. Main block is mandatory. There is no need for main block to be inside begin..end.

It could be a single statement. The script should set the result in a special variable called "Output".

Examples:

SCRIPT 1:

```
//This is a function declaration
function GetNameAndState : String;
begin
  Result := GetFieldValue('Name') + GetFieldValue('State');
end;

Output := GetNameAndState; //This is main block
```

SCRIPT 2:

```
//This is a procedure declaration
procedure SetNameAndState;
begin
  Output := GetFieldValue('Name') + GetFieldValue('State');
end;

SetNameAndState; //This is main block
```

SCRIPT 2:

```
//This script has only main block
Output := GetFieldValue('Name') + GetFieldValue('State');
```

Like in Pascal, statements should be terminated by ";" character. Begin..end blocks are allowed to group statements.

NOTE	The script must set the result in the implicitly declared variable "Output" to display it in the calculated field.

Identifiers

Identifier names in script (variable names, function and procedure names, etc.) follow the most common rules in Pascal: should begin with a character (a..z or A..Z), or '_', and can be followed by alphanumeric chars or '_' char. Cannot contain any other character or spaces.

Valid identifiers:

```
VarName
_Some
V1A2
_____Some____
```

Invalid identifiers:

```
2Var
My Name
Some-more This,is,not,valid
```

Assign statements

Just like in Pascal, assign statements (assign a value or expression result to a variable or object property) are built using ":=".

Examples:

```
MyVar := 2;
Caption := 'This ' + 'is ok.';
```

Character strings

Strings (sequence of characters) are declared in Pascal using single quote (') character. Double quotes (") are not used. You can also use #nn to declare a character inside a string. There is no need to use '+' operator to add a character to a string.
Examples:

```
A  := 'This is a text';
Str:= 'Text '+'concat';
```

```
B  := 'String with CR and LF char at the end'#13#10;
C  := 'String with '#33#34' characters in the middle';
```

Comments

Comments can be inserted inside script. You can use // chars or (*   *) or { } blocks. Using // char the comment will finish at the end of line.

```
//This is a single line comment

(* This is another comment *)

{
  And this is a comment
  with two lines
}
```

Variables

There is no need to declare variable types in script. Thus, you declare variable just using var directive and its name. There is no need to declare variables if scripter property OptionExplicit is set to false. In this case, variables are implicit declared. If you want to have more control over the script, set OptionExplicit property to true. This will raise a compile error if variable is used but not declared in script.

Examples:

SCRIPT 1:

```
procedure Msg;
var S;
begin
   S:='Hello world!';
   ShowMessage(S);
end;
```

SCRIPT 2:

```
var A;
begin
 A:=0;
 A:=A+1;
end;
```

SCRIPT 3:

```
var S;
S:='Hello World!';
ShowMessage(S);
```

Indexes

Strings, arrays and array properties can be indexed using "[" and "]" chars. For example, if Str is a string variable, the expression Str[3] returns the third character in the string denoted by Str, while Str[I + 1] returns the character immediately after the one indexed by I.
Examples:

```
MyChar:=MyStr[2];
MyStr[1]:='A';
MyArray[1,2]:=1530;
Lines.Strings[2]:='Some text';
```

Arrays

Script support array constructors and support to variant arrays. To construct an array, use "[" and "]" chars. You can construct multi-index array nesting array constructors. You can then access arrays using indexes. If array is multi-index, separate indexes using ",".
If variable is a variant array, script automatically support indexing in that variable. A variable is a variant array is it was assigned using an array constructor, if it is a direct reference to a Delphi variable which is a variant array (see Delphi integration later) or if it was created using VarArrayCreate procedure. Arrays in script are 0-based index.
Examples:

```
NewArray := [ 2,4,6,8 ];
Num:=NewArray[1]; //Num receives "4"
MultiArray := [ ['green','red','blue'] , ['apple','orange','lemon']
];
Str:=MultiArray[0,2]; //Str receives 'blue'
MultiArray[1,1]:='new orange';
```

If statements

There are two forms of if statement: if...then and if...then...else. Like normal Pascal, if the if expression is true, the statement (or block) is executed. If there is else part and expression is false, statement (or block) after else is execute.

Examples:

```
if J <> 0 then Output:= I/J;
if J = 0 then Exit else Output := I/J;
if J <> 0 then
begin
   Output:= I/J;
   Count := Count + 1;
end
else
   Done := True;
```

While statements

A while statement is used to repeat a statement or a block, while a control condition (expression) is evaluated as true. The control condition is evaluated before the statement. Hence, if the control condition is false at first iteration, the statement sequence is never executed. The while statement executes its constituent statement (or block) repeatedly, testing expression before each iteration. As long as expression returns true, execution continues.

Examples:

```
while Data[I] <> X do
I := I + 1;

while I > 0 do
begin
   if Odd(I) then Z := Z * X;
   I := I div 2;
   X := Sqr(X);
end;
```

Repeat statements

The syntax of a repeat statement is *repeat statement1; ...; statementn; until expression* where expression returns a Boolean value. The repeat statement executes its sequence of constituent statements continually, testing expression after each iteration. When expression returns true, the repeat statement terminates. The sequence is always executed at least once because expression is not evaluated until after the first iteration.

Examples:

```
repeat
   K := I mod J;
   I := J;
   J := K;
until J = 0;
```

For statements

Scripter support for statements with the following syntax:

```
for counter := initialValue to finalValue do statement
```

For statement set counter to initialValue, repeats execution of statement (or block) and increment value of counter until counter reaches finalValue.

Examples:

SCRIPT 1:

```
for c:=1 to 10 do
   a:=a+c;
```

SCRIPT 2:

```
for i:=a to b do
begin
    j:=i^2;
   sum:=sum+j;
end;
```

Case statements

Scripter support case statements with following syntax:

```
 case selectorExpression of
   caseexpr1: statement1;
    ...
   caseexprn: statementn;
  else
      elsestatement;
end
```

if selectorExpression matches the result of one of caseexprn expressions, the respective statement (or block) will be execute. Otherwise, elsestatement will be execute. Else part of case statement is optional. Different from Delphi, case statement in script doesn't need to use only ordinal values. You can use expressions of any type in both selector expression and case expression.

Example:

```
case uppercase(Fruit) of
   'lime': color :='green';
   'orange': color := 'orange';
   'apple': color := 'red';
else
   color := 'black';
```

```
end;
```

Function and procedure declaration

Declaration of functions and procedures are similar to Object Pascal in Delphi, with the difference you don't specify variable types. Just like OP, to return function values, use implicit declared result variable. Parameters by reference can also be used, with the restriction mentioned: no need to specify variable types.

Examples:

```
procedure HelloWorld;
begin
   ShowMessage('Hello world!');
end;

procedure UpcaseMessage(Msg);
begin
   ShowMessage(Uppercase(Msg));
end;

function TodayAsString;
begin
    result:=DateToStr(Date);
end;

function Max(A,B);
begin
    if A>B then
    result:=A
else
    result:=B;
end;

procedure SwapValues(var A, B);
Var
```

```
  Temp;
begin
    Temp:=A;
    A:=B;
    B:=Temp;
end;
```

Functions

Functions are written in both Pascal and Basic syntax. However, there are some functions that can be used in Pascal as well as Basic scripts. Click the below links to view the listed functions for each:

- **Functions available in Pascal and Basic Languages**
- **Functions available in Basic Language only**

Functions available in Pascal and Basic Languages

NOTE	Although the signature of the functions described below are written in Pascal syntax they can be used in Pascal as well as Basic scripts. When using these functions in the Basic script you need to supply corresponding data types of the Basic language as parameters.

Functions to access field value

Name	Description

GetFieldValue	`function GetFieldValue(FieldName : String) : Variant;`
	
	Returns the value in a field, such as State, Priority, Name, etc., of the current record. Paremeter FieldName should have the name of the field whose value you want.
	Example:
	`Output := GetFieldValue('Est. Effort') - GetFieldValue('Actual Effort');`
	
	NOTE: You can only get value of following type of fields: Number, Decimal, Text, True/False, Date, Time, List, Project, User, Team Member and Multi-Value.
	This function cannot be used to get value of the following fields:

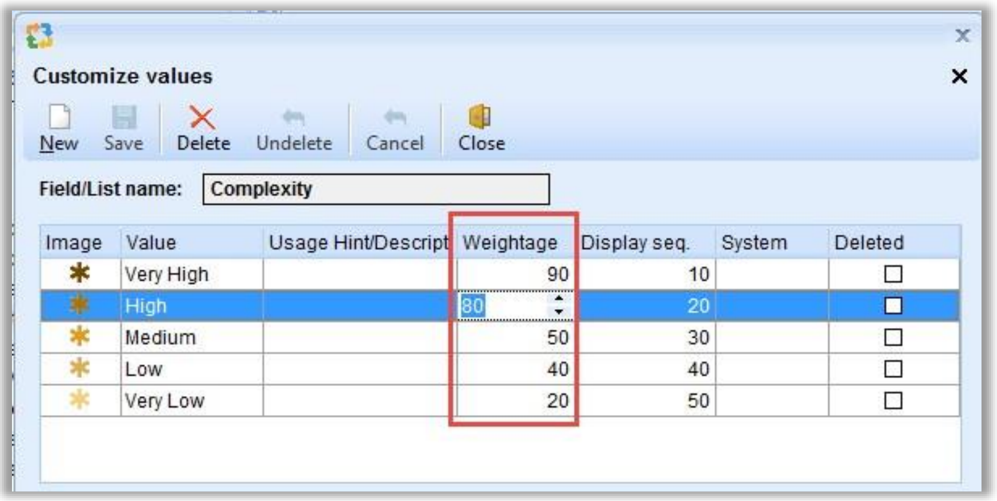
		1. Value from a Large Field such as Large Text, Rich Text, OLE Object, Diagrams, Use Case Flows, Test Case Steps, etc. 2. Value from another Calculated Field.

GetWeightageOf

```
function GetWeightageOf(FieldName : String) : Integer;
```

This function returns the weightage of the value in a List field.

When you define allowed values for a List field, you can define weightage of each value. Refer to the screenshot below. You can use this function in the calculation formula to get the weightage of the value of a List field.



Field/List name: Complexity

Image	Value	Usage Hint/Descript	Weightage	Display seq.	System	Deleted
✱	Very High		90	10		☐
✱	High		80	20		☐
✱	Medium		50	30		☐
✱	Low		40	40		☐
✱	Very Low		20	50		☐

Example:

```
Output := GetWeightageOf('Complexity');
```

If the value in Complexity field is Very High, the calculation formula will display 90, if the value is High the calculation formula will display 80 and so on.

IsFieldNull	`function IsFieldNull(FieldName : String) : Boolean;`
	
	This function can be used to detect whether a field is empty. The function will return True if there is no value specified in the field, otherwise it will return False.
	
	Example:
	`if not IsFieldNull('Description') then Output := 'Description is empty' else`
	`  Output := 'Description has value';`
	
	NOTE — Unlike **GetFieldValueOf**, you can send the name of large fields such as Rich Text, Large Text, OLE Objects, etc. as a parameter in this function.

Arithmetic Functions

Name	Description
Abs	`function Abs(X);` Abs returns the absolute value of the argument, X. X is an integer-type or real-type expression.

Dec	`procedure Dec(var X[ ; N: Longint]);`
	
	The Dec procedure subtracts one or N from a variable.
	
	X is a variable of an ordinal type (including Int64), or a pointer type if the extended syntax is enabled.
	N is an integer-type expression.
	X decrements by 1, or by N if N is specified; that is, Dec(X) corresponds to the statement X := X - 1, and Dec(X, N) corresponds to the statement X := X - N. However, Dec generates optimized code and is especially useful in tight loops.
	
	NOTE — If X is a pointer type, it decrements X by N times the size of the type pointed to. Thus, given type
	`  PMytype = ^TMyType;`
	`and var`
	`  P: PMyType; the statement Dec(P); decrements P by SizeOf(TMyType).`
	
	CAUTION — Do NOT use Dec on ordinal-type properties, if the property uses a write procedure.

Frac	`function Frac(X: Extended): Extended;` The Frac function returns the fractional part of the argument X. X is a real-type expression. The result is the fractional part of X; that is, Frac(X) = X - Int(X).

Inc

```
procedure Inc(var X [ ; N: Longint ] );
```

Inc adds one or N to the variable X.

X is a variable of an ordinal type (including Int64), or a pointer type if the extended syntax is enabled.
N is an integer-type expression.
X increments by 1, or by N if N is specified; that is, Inc(X) corresponds to the statement X := X + 1, and Inc(X, N) corresponds to the statement X := X + N. However, Inc generates optimized code and is especially useful in tight loops.

NOTE	If X is a pointer type, it increments X by N times the size of the type pointed to. Thus, given type `PMytype = ^TMyType;` `and var` `P: PMyType; the statement Inc(P); increments P by SizeOf(TMyType).`

CAUTION	Do not use Inc on properties.

NOTE	Inc(S, I) where S is a ShortInt and I is a number greater than 127 will cause an EIntOverFlow exception to be raised if range and overflow checking are on. Under Delphi 1.0 this did not raise an exception.

Int
Int returns the integer part of X; that is, X rounded toward zero. X is a real-type expression.

IntToHex	`function IntToHex(Value: Integer; Digits: Integer): string; overload;`
	`function IntToHex(Value: Int64; Digits: Integer): string; overload;`
	
	IntToHex converts a number into a string containing the number's hexadecimal (base 16) representation. Value is the number to convert. Digits indicates the minimum number of hexadecimal digits to return.

Odd	`function Odd(X: Longint): Boolean;` Odd tests if the argument is an odd number. It returns True if X is an odd number, False if X is even.
Random	`function Random [ ( Range: Integer) ];` Random returns a random number within the range 0 <= X < Range. If Range is not specified, the result is a realtype random number within the range 0 <= X < 1. To initialize the random number generator, add a single call Randomize or assign a value to the RandSeed variable before making any calls to Random. **NOTE** Because the implementation of the Random function may change between compiler versions, we do not recommend using Random for encryption or other purposes that require reproducible sequences of pseudorandom numbers.
Round	`function Round(X: Extended): Int64;` The Round function rounds a real-type value to an integer-type value. X is a real-type expression. Round returns an Int64 value that is the value of X rounded to the nearest whole number. If X is exactly halfway between two whole numbers, the result is always the even number. This method of rounding is often called “Banker’s Rounding”. If the rounded value of X is not within the Int64 range, a run-time error is generated, which can be handled using the EInvalidOp exception.

	NOTE The behavior of Round can be affected by the Set8087CW procedure or SetRoundMode function.

Sqr	`function Sqr(X: Extended): Extended;`
	`function Sqr(X: Integer): Integer;`
	
	The Sqr function returns the square of the argument.
	X is a floating-point expression. The result, of the same type as X, is the square of X, or X*X.

Sqrt	`function Sqrt(X: Extended): Extended;` X is a floating-point expression. The result is the square root of X.
Trunc	`function Trunc(X: Extended): Int64;` The Trunc function truncates a real-type value to an integer-type value. X is a real-type expression. Trunc returns an Int64 value that is the value of X rounded toward zero. If the truncated value of X is not within the Int64 range, an EInvalidOp exception is raised.

String manipulation functions

Name	Description

AnsiCompareStr	`function AnsiCompareStr(const S1, S2: string): Integer;`
	
	AnsiCompareStr compares S1 to S2, with case sensitivity. The compare operation is controlled by the current locale. The return value is: Condition Return Value
	
	`S1 > S2    > 0`
	`S1 < S2    < 0`
	`S1 = S2    = 0`
	
	NOTE Most locales consider lowercase characters to be less than the corresponding uppercase characters. This is in contrast to ASCII order, in which lowercase characters are greater than uppercase characters.
	Thus,
	AnsiCompareStr('a','A') returns a value less than zero, while CompareStr('a','A') returns a value greater than zero.

AnsiCompareText	`function AnsiCompareText(const S1, S2: string): Integer;` AnsiCompareText compares S1 to S2, without case sensitivity. The compare operation is controlled by the current locale. AnsiCompareText returns a value less than 0 if S1 < S2, a value greater than 0 if S1 > S2, and returns 0 if S1 = S2.
AnsiLowerCase	`function AnsiLowerCase(const S: string): string;` AnsiLowerCase returns a string that is a copy of the given string converted to lower case. The conversion uses the current locale. **NOTE** This function supports multi-byte character sets (MBCS).
Append	`procedure Append(var F: Text);`

	Call Append to ensure that a file is opened with write-only access with the file pointer positioned at the end of the file. F is a text file variable and must be associated with an external file using AssignFile. If no external file of the given name exists, an error occurs. If F is already open, it is closed, then reopened. The current file position is set to the end of the file. If a Ctrl+Z (ASCII 26) is present in the last 128-byte block of the file, the current file position is set so that the next character added to the file overwrites the first Ctrl+Z in the block. In this way, text can be appended to a file that terminates with a Ctrl+Z. If F was not assigned a name, then, after the call to Append, F refers to the standard output file.
CompareStr	`function CompareStr(const S1, S2: string): Integer;` CompareStr compares S1 to S2, with case-sensitivity. The return value is less than 0 if S1 is less than S2, 0 if S1 equals S2, or greater than 0 if S1 is greater than S2. The compare operation is based on the 8-bit ordinal value of each character and is not affected by the current locale.
CompareText	`function CompareText(const S1, S2: string): Integer;` CompareText compares S1 and S2 and returns 0 if they are equal. If S1 is greater than S2, CompareText returns an integer greater than 0. If S1 is less than S2, CompareText returns an integer less than 0. CompareText is not case sensitive and is not affected by the current locale.

Copy	`function Copy(S; Index, Count: Integer): string;`
	`function Copy(S; Index, Count: Integer): array;`
	
	S is an expression of a string or dynamic-array type. Index and Count are integer-type expressions. Copy returns a substring or sub array containing Count characters or elements starting at S[Index].
	
	If Index is larger than the length of S, Copy returns an empty string or array.

	If Count specifies more characters or array elements than are available, only the characters or elements from S[Index] to the end of S are returned. **NOTE** When S is a dynamic array, Copy can only be used as a parameter in a call to a procedure or function that expects an array parameter. That is, it acts like the Slice function when working with dynamic arrays.
Delete	`procedure Delete(var S: string; Index, Count:Integer);` Delete removes a substring of Count characters from string S starting with S[Index]. S is a string-type variable. Index and Count are integer-type expressions. If index is larger than the length of the S or less than 1, no characters are deleted. If count specifies more characters than remain starting at the index, Delete removes the rest of the string. If count is less than 0, no characters are deleted.
Delete	`procedure Delete(var S: string; Index, Count:Integer);` Delete removes a substring of Count characters from string S starting with S[Index]. S is a string-type variable. Index and Count are integer-type expressions. If index is larger than the length of the S or less than 1, no characters are deleted. If count specifies more characters than remain starting at the index, Delete removes the rest of the string. If count is less than 0, no characters are deleted.
FloatToStr	`function FloatToStr(Value: Extended): string;` FloatToStr converts the floating-point value given by Value to its string representation. The conversion uses general number format with 15 significant digits. For greater control over the formatting of the string, use the FloatToStrF function.
Format	`function Format(const Format: string; const Args: array of const): string;`

	This function formats the series of arguments in the open array Args. Formatting is controlled by the format string Format; the results are returned in the function result as a string.
FormatFloat	`function FormatFloat(const Format: string; Value: Extended): string;` FormatFloat formats the floating-point value given by Value using the format string given by Format. The following format specifiers are supported in the format string: **Specifier - Represents** **0 - Digit place holder** If the value being formatted has a digit in the position where the '0' appears in the format string, then that digit is copied to the output string. Otherwise, a '0' is stored in that position in the output string. **# - Digit placeholder** If the value being formatted has a digit in the position where the '#' appears in the format string, then that digit is copied to the output string. Otherwise, nothing is stored in that position in the output string. **. - Decimal point** The first '.' character in the format string determines the location of the decimal separator in the formatted value; any additional '.' characters are ignored. The actual character used as a decimal separator in the output string is determined by the DecimalSeparator global variable. The default value of DecimalSeparator is specified in the Number Format of the International section in the Windows Control Panel. **, - Thousand separator** If the format string contains one or more ',' characters, the output will have thousand separators inserted between each group of three digits to the left of the decimal point. The placement and number of ',' characters in the format string does not affect the output, except to indicate that thousand separators are wanted. The actual character used as a thousand separator in the output is determined by the ThousandSeparator global variable. The default value of ThousandSeparator is specified in the Number Format of the International section in the Windows Control Panel.

	E+ - Scientific notation If any of the strings 'E+', 'E-', 'e+', or 'e-' are contained in the format string, the number is formatted using scientific notation. A group of up to four '0' characters can immediately follow the 'E+', 'E-', 'e+', or 'e-' to determine the minimum number of digits in the exponent. The 'E+' and 'e+' formats cause a plus sign to be output for positive exponents and a minus sign to be output for negative exponents. The 'E-' and 'e-' formats output a sign character only for negative exponents. **'xx'/"xx" - Characters** enclosed in single or double quotes are output as-is, and do not affect formatting. **; - Separates sections** for positive, negative, and zero numbers in the format string. The locations of the leftmost '0' before the decimal point in the format string and the rightmost '0' after the decimal point in the format string determine the range of digits that are always present in the output string. The number being formatted is always rounded to as many decimal places as there are digit placeholders ('0' or '#') to the right of the decimal point. If the format string contains no decimal point, the value being formatted is rounded to the nearest whole number. If the number being formatted has more digits to the left of the decimal separator than there are digit placeholders to the left of the '.' character in the format string, the extra digits are output before the first digit placeholder. To allow different formats for positive, negative, and zero values, the format string can contain between one and three sections separated by semicolons. • **One section:** The format string applies to all values. • **Two sections:** The first section applies to positive values and zeros, and the second section applies to negative values. • **Three sections:** The first section applies to positive values, the second applies to negative values, and the third applies to zeros. If the section for negative values or the section for zero values is empty, that is if there is nothing between the semicolons that delimit the section, the section for positive values is used instead. If the section for positive values is empty, or if the entire format string is empty, the value is formatted using general floating-point formatting with 15 significant digits, corresponding to a call to FloatToStrF with the ffGeneral format. General

	floating-point formatting is also used if the value has more than 18 digits to the left of the decimal point and the format string does not specify scientific notation.
Insert	`procedure Insert(Source: string; var S: string; Index: Integer);` Insert merges Source into S at the position S[index]. Source is a string-type expression. S is a string-type variable of any length. Index is an integer-type expression. It is a character index and not a byte index. If Index is less than 1, it is mapped to a 1. If it is past the end of the string, it is set to the length of the string, turning the operation into an append. If the Source parameter is an empty string, Insert does nothing. Insert throws an EOutOfMemory exception, if it is unable to allocate enough memory to accomodate the new returned string.

IntToStr	`function IntToStr(Value: Integer): string; overload;`
	`function IntToStr(Value: Int64): string; overload;`
	
	IntToStr converts an integer into a string containing the decimal representation of that number.

Length	`function Length(S): Integer;` Length returns the number of characters actually used in the string or the number of elements in the array. For single-byte (AnsiString) and multibyte strings, Length returns the number of bytes used by the string. For Unicode (WideString) strings, Length returns the number of bytes divided by two. S is a string- or array-valued expression.
LowerCase	`0function LowerCase(const S: string): string;` LowerCase returns a string with the same text as the string passed in S, but with all letters converted to lowercase. The conversion affects only 7-bit ASCII characters between 'A' and 'Z'. To convert 8-bit international characters, use AnsiLowerCase.

Pos	`function Pos(Substr: string; S: string): Integer;` Pos searches for a substring, Substr, in a string, S. Substr and S are string-type expressions. Pos searches for Substr within S and returns an integer value that is the index of the first character of Substr within S. Pos is case-sensitive. If Substr is not found, Pos returns zero.

Trim	`function Trim(const S: string): string; overload;`
	`function Trim(const S: WideString): WideString; overload;`
	
	Trim removes leading and trailing spaces and control characters from the given string S.

TrimLeft	`function TrimLeft(const S: string): string; overload;`
	`function TrimLeft(const S: WideString): WideString; overload;`
	
	TrimLeft returns a copy of the string S with leading spaces and control characters removed.

TrimRight	`function TrimRight(const S: string): string; overload;`
	`function TrimRight(const S: WideString): WideString; overload;`
	
	TrimRight returns a copy of the string S with trailing spaces and control characters removed.

UpperCase	`function UpperCase(const S: string): string;` UpperCase returns a copy of the string S, with the same text but with all 7-bit ASCII characters between 'a' and 'z' converted to uppercase. To convert 8-bit international characters, use AnsiUpperCase instead.

Trigonometric functions

Name	**Description**

ArcTan	`function ArcTan(X: Extended): Extended;` ArcTan returns the arctangent of X.
Cos	`function Cos(X: Extended): Extended;` Cos returns the cosine of the angle X, in radians.
Sin	`function Sin(X: Extended): Extended;` The Sin function returns the sine of the argument. X is a real-type expression. Sin returns the sine of the angle X in radians.

Date and Time functions

Name	**Description**
Date	`function Date: TDateTime;` Use Date to obtain the current local date as a TDateTime value. The time portion of the value is 0 (midnight).
DateTimeToStr	`function DateTimeToStr(DateTime: TDateTime): string;` DateTimeToString converts the TDateTime value given by DateTime using the format given by the ShortDateFormat global variable, followed by the time using the format given by the LongTimeFormat global variable. The time is not displayed if the fractional part of the DateTime value is zero.
DateToStr	`function DateToStr(Date: TDateTime): string;` Use DateToStr to obtain a string representation of a date value that can be used for display purposes. The conversion uses the format specified by the ShortDateFormat global variable.

DayOfWeek	`function DayOfWeek(Date: TDateTime): Integer;` DayOfWeek returns the day of the week of the specified date as an integer between 1 and 7, where Sunday is the first day of the week and Saturday is the seventh. **NOTE** DayOfWeek is not compliant with the ISO 8601 standard, which defines Monday as the first day of the week. For an ISO 8601 compliant version, use the DayOfTheWeek function instead.
DecodeDate	`procedure DecodeDate(Date: TDateTime; var Year, Month, Day: Word);` The DecodeDate procedure breaks the value specified as the Date parameter into Year, Month, and Day values. If the given TDateTime value has a negative (BC) year, the year, month, and day return parameters are all set to zero.
DecodeTime	`procedure DecodeTime(Time: TDateTime; var Hour, Min, Sec, MSec: Word);` DecodeTime breaks the object specified as the Time parameter into hours, minutes, seconds, and milliseconds.

EncodeDate	`function EncodeDate(Year, Month, Day: Word): TDateTime;`
	`function TryEncodeDate(Year, Month, Day: Word; out Date: TDateTime): Boolean;`
	
	EncodeDate returns a TDateTime value from the values specified as the Year, Month, and Day parameters.
	
	The year must be between 1 and 9999.
	Valid Month values are 1 through 12.
	Valid Day values are 1 through 28, 29, 30, or 31, depending on the Month value.
	
	For example,
	the possible Day values for month 2 (February) are 1 through 28 or 1 through 29, depending on whether or not the Year value specifies a leap year.

	If the specified values are not within range, EncodeDate raises an EConvertError exception. TryEncodeDate is identical to EncodeDate, except that TryEncodeDate responds to out of range parameters by returning False instead of raising an exception.

EncodeTime	`function EncodeTime(Hour, Min, Sec, MSec: Word): TDateTime;`
	`function TryEncodeTime(Hour, Min, Sec, MSec: Word; out Time: TDateTime): Boolean;`
	
	EncodeTime encodes the given hour, minute, second, and millisecond into a TDateTime value.
	
	Valid Hour values are 0 through 24. If Hour is 24, Min, Sec, and MSec must all be 0, and the resulting TDateTime value represents midnight (12:00:00:000 AM) of the following day.
	Valid Min and Sec values are 0 through 59.
	Valid MSec values are 0 through 999.
	
	If the specified values are not within range, EncodeTime raises an EConvertError exception.
	The resulting value is a number between 0 and 1 (inclusive) that indicates the fractional part of a day given by the specified time or (if 1.0) midnight on the following day. The value 0 corresponds to midnight, 0.5 corresponds to noon, 0.75 corresponds to 6:00 pm, and so on.
	
	TryEncodeTime is identical to EncodeTime, except that TryEncodeTime responds to out of range parameters by returning False instead of raising an exception.

FormatDateTime	`function FormatDateTime(const Format: string; DateTime: TDateTime): string;` FormatDateTime formats the TDateTime value given by DateTime using the format given by Format. If the string specified by the Format parameter is empty, the TDateTime value is formatted as if a 'c' format specifier had been given.

StrToDate

```
function StrToDate(const S: string): TDateTime;
```

Call StrToDate to parse a string that specifies a date. If S does not contain a valid date, StrToDate raises an EConvertError exception.

S must consist of two or three numbers, separated by the character defined by the DateSeparator global variable. The order for month, day, and year is determined by the ShortDateFormat global variable--possible combinations are m/d/y, d/m/y, and y/m/d.

If S contains only two numbers, it is interpreted as a date (m/d or d/m) in the current year.

Year values between 0 and 99 are converted using the TwoDigitYearCenturyWindow global variable.

If TwoDigitYearCenturyWindow is 0, year values between 0 and 99 are assumed to be in the current century.
If TwoDigitYearCenturyWindow is greater than 0, its value is subtracted from the current year to determine the "pivot"; years on or after the pivot are kept in the current century, while years prior to the pivot are moved to the next century.

For example:

Current year	**TwoDigitYearCenturyWindow**	**Pivot**	**date = mm/dd/03**	**date = mm/dd/50**	**date = mm/dd/68**
1998	0	1900	1903	1950	1968
2002	0	2000	2003	2050	2068
1998	50	1948	2003	1950	1968
2000	50	1950	2003	1950	1968
2002	50	1952	2003	2050	1968
2020	50	1970	2003	2050	2068
2020	10	2010	2103	2050	2068

	NOTE The format of the date string varies when the values of date/time formatting variables are changed.
StrToDateTime	`function StrToDateTime(const S: string): TDateTime;` Call StrToDate to parse a string that specifies a date and time value. If S does not contain a valid date, StrToDate raises an EConvertError exception. The S parameter must use the current locale’s date/time format. In the US, this is commonly MM/DD/YY HH:MM:SS format. Specifying AM or PM as part of the time is optional, as are the seconds. Use 24-hour time (7:45 PM is entered as 19:45, for example) if AM or PM is not specified. Y2K issue: The conversion of two-digit year values is determined by the TwoDigitYearCenturyWindow variable. **NOTE** The format of the date and time string varies when the values of date/time formatting variables are changed.
StrToFloat	`function StrToFloat(const S: string): Extended;` Use StrToFloat to convert astring, S, to a floating-point value. S must consist of an optional sign (+ or -), a string of digits with an optional decimal point, and an optional mantissa. The mantissa consists of 'E' or 'e' followed by an optional sign (+ or -) and a whole number. Leading and trailing blanks are ignored. The DecimalSeparator global variable defines the character that must be used as a decimal point. Thousand separators and currency symbols are not allowed in the string. If S doesn't contain a valid value, StrToFloat raises an EConvertError exception.

StrToInt	`function StrToInt(const S: string): Integer;` StrToInt converts the string S, which represents an integer-type number in either decimal or hexadecimal notation, into a number. If S does not represent a valid number, StrToInt raises an EConvertError exception.
StrToIntDef	`function StrToIntDef(const S: string; Default: Integer): Integer;` StrToIntDef converts the string S, which represents an integer-type number in either decimal or hexadecimal notation, into a number. If S does not represent a valid number, StrToIntDef returns the number passed in Default.
StrToTime	`function StrToTime(const S: string): TDateTime;` Call StrToTime to parse a string that specifies a time value. If S does not contain a valid time, StrToTime raises an EConvertError exception. The S parameter must consist of two or three numbers, separated by the character defined by the TimeSeparator global variable, optionally followed by an AM or PM indicator. The numbers represent hour, minute, and (optionally) second, in that order. If the time is followed by AM or PM, it is assumed to be in 12-hour clock format. If no AM or PM indicator is included, the time is assumed to be in 24-hour clock format. **NOTE** The format of the date and time string varies when the values of date/time formatting variables are changed.
Time	`function Time: TDateTime;` Time returns the current time as a TDateTime value.

TimeToStr	`function TimeToStr(Time: TDateTime): string;` TimeToStr converts the Time parameter, a TDateTime value, to a string. The conversion uses the format specified by the LongTimeFormat global variable. Change the format of the string by changing the values of some of the date and time variables.
IncMonth	`function IncMonth(const Date: TDateTime; NumberOfMonths: Integer = 1): TDateTime;` IncMonth returns the value of the Date parameter, incremented by NumberOfMonths months. NumberOfMonths can be negative, to return a date N months previous. If the input day of month is greater than the last day of the resulting month, the day is set to the last day of the resulting month. The time of day specified by the Date parameter is copied to the result.
IsLeapYear	`function IsLeapYear(Year: Word): Boolean;` Call IsLeapYear to determine whether the year specified by the Year parameter is a leap year. Year specifies the calendar year. Use YearOf to obtain the value of Year for IsLeapYear from a TDateTime value.
Now	`function Now: TDateTime;` Returns the current date and time, corresponding to Date + Time.

Logarithm functions

Name	**Description**
Exp	`function Exp(X: Real): Real;` Exp returns the value of e raised to the power of X, where e is the base of the natural logarithms.

Ln	`function Ln(X: Real): Real;` Ln returns the natural logarithm (Ln(e) = 1) of the real-type expression X.

Miscellaneous functions

Name	Description
Assigned	`function Assigned(const P): Boolean;` Use Assigned to determine whether the pointer or procedure referenced by P is nil. P must be a variable reference of a pointer or procedural type. Assigned(P) corresponds to the test P<> nil for a pointer variable, and @P <> nil for a procedural variable. Assigned returns False if P is nil, True otherwise. **NOTE** Assigned cannot detect a dangling pointer--that is, one that isn't nil but no longer points to valid data. For example, in the code example for Assigned, Assigned won't detect the fact that P isn't valid.
Chr	`function Chr(X: Byte): Char;` Chr returns the character with the ordinal value (ASCII value) of the byte-type expression, X.

Eof	`function Eof(var F): Boolean;`
	`function Eof [ (var F: Text) ]: Boolean;`
	
	Eof tests whether the current file position is the end-of-file.
	
	F is a file variable that has been opened for reading. If F is omitted, the standard file variable Input is assumed.
	Eof(F) returns True if the current file position is beyond the last character of the file or if the file is empty; otherwise, Eof(F) returns False.
	
	NOTE Eof fails if the file F has been opened in write-only mode. For example, you can't use Eof with files opened using the Append or Rewrite, which open a file in write-only mode.

High	`function High(X);` Call High to obtain the upper limit of an Ordinal, Array, or string value. The result type is X, or the index type of X. X is either a type identifier or a variable reference. The type denoted by X, or the type of the variable denoted by X, must be one of the following: **For this type - High returns** Ordinal type (includes Int64) - The highest value in the range of the type. Array type - The highest value within the range of the index type of the array. For empty arrays, High returns –1. String type - The declared size of the string. Open array - The value, of type Integer, giving the number of elements in the actual parameter minus one. String parameter - The value, of type Integer, giving the number of elements in the actual parameter minus one.
IsValidIdent	`function IsValidIdent(const Ident: string): Boolean;` IsValidIdent returns true if the given string is a valid identifier.

	An identifier is defined as a character from the set ['A'..'Z', 'a'..'z', '_'] followed by zero or more characters from the set ['A'..'Z', 'a'..'z', '0..'9', '_'].
Low	`function Low(X);` Call Low to obtain the lowest value or first element of an Ordinal, Array or string. Result type is X, or the index type of X where X is either a type identifier or a variable reference. **Type - Low returns** Ordinal type (includes Int64) - The lowest value in the range of the type Array type - The lowest value within the range of the index type of the array String type - eturns 0 only on shortstrings Open array - Returns 0 String parameter - Returns 0
Ord	`function Ord(X);` X is an ordinal-type expression. The result is the ordinal position of X; its type is the smallest standard integer type that can hold all values of X's type. Ord cannot operate on Int64 values.
VarArrayCreate	`function VarArrayCreate(const Bounds: array of Integer; VarType: TVarType): Variant;` VarArrayCreate creates a variant array with the bounds given by Bounds and the element type given by VarType. The Bounds parameter must contain an even number of values, where each pair of values specifies the upper and lower bounds of one dimension of the array. The element type of the array, given by the VarType parameter, is a variant type code. This must be one of the constants defined in the System unit. It cannot include the varArray or varByRef bits. The element type cannot be varString or a custom Variant type. To create a variant array of strings use the varOleStr type code. If the element type is varVariant, the elements of the array are themselves variants and can in turn contain variant arrays.

	NOTE Variant arrays with an element type of varByte are the preferred method of passing binary data between OLE Automation controllers and servers. Such arrays are subject to no translation of their data, and can be efficiently accessed using the VarArrayLock and VarArrayUnlock routines.
VarArrayHighBound	`function VarArrayHighBound(const A: Variant; Dim: Integer): Integer;` VarArrayHighBound returns the high bound of the given dimension in the given variant array. The Dim parameter should be 1 for the leftmost dimension, 2 for the second leftmost dimension, and so on. An EVariantError exception is raised if the variant given by A is not an array, or if the dimension specified by Dim does not exist.
VarArrayLowBound	`function VarArrayLowBound(const A: Variant; Dim: Integer): Integer` VarArrayLowBound returns the low bound of the given dimension in the given variant array. The Dim parameter should be 1 for the leftmost dimension, 2 for the second leftmost dimension, and so on. An EVariantError exception is raised if the variant given by A is not an array, or if the dimension specified by Dim does not exist.
VarIsNull	`function VarIsNull(const V: Variant): Boolean;` VarIsNull returns True if the given variant contains the value Null. If the variant contains any other value, the function result is False. **NOTE** Do not confuse a Null variant with an unassigned variant. A Null variant is still assigned, but has the value Null. Unlike unassigned variants, Null variants can be used in expressions and can be converted to other types of variants.

VarToStr	`function VarToStr(const V: Variant): string;` VarToStr converts the data in the variant V to a string and returns the result. If the variant has a null value, VarToStr returns an empty string.

Functions available in Basic language only

Name	**Description**

Asc	Returns an Integer value that represents the character code corresponding to a character.
	
	`Public Overloads Function Asc(ByVal String As Char) As Integer`
	`Public Overloads Function AscW(ByVal String As Char) As Integer`
	`' -or-`
	`Public Overloads Function Asc(ByVal String As String) As Integer`
	`Public Overloads Function AscW(ByVal String As String) As Integer`
	
	For more information, refer to Asc, AscW Functions.

Atn	Returns a Double specifying the arctangent of a number. For more information, refer to Atn Function.
CBool	Returns an expression that has been converted to a Variant of subtype Boolean. `CBool(expression)` For more information, refer to CBool Function.
CByte	Returns an expression that has been converted to a Variant of subtype Byte.

	`CByte(expression)` For more information, refer to CByte Function.
CCur	Returns an expression that has been converted to a Variant of subtype Currency. `CCur(expression)` For more information, refer to CCur Function.
CDate	Returns an expression that has been converted to a Variant of subtype Date. `CDate(date)` For more information, refer to CDate Function.
CDbl	Returns an expression that has been converted to a Variant of subtype Double. `CDbl(expression)` For more information, refer to CDbl Function.
Cint	Returns an expression that has been converted to a Variant of subtype Integer. `CInt(expression)` For more information, refer to CInt Function.
CLng	Returns an expression that has been converted to a Variant of subtype Long. `CLng(expression)` For more information, refer to CLng Function.
CSng	Returns an expression that has been converted to a Variant of subtype Single.

	`CSng(expression)` For more information, refer to CSng Function.
CStr	Returns an expression that has been converted to a Variant of subtype String. `CStr(expression)` For more information, refer to CStr Function.
DatePart	Returns the specified part of a given date. `DatePart(interval, date[, firstdayofweek[, firstweekofyear]])` For more information, refer to DatePart Function.
DateSerial	Returns a Variant of subtype Date for a specified year, month, and day. `DateSerial(year, month, day)` For more information, refer to DateSerial Function.
DateValue	Returns a Variant of subtype Date. `DateValue(date)` For more information, refer to DateValue Function.
Day	Returns a whole number between 1 and 31, inclusive, representing the day of the month. `Day(date)` For more information, refer to Day Function.
Fix	Returns the integer portion of a number.

	`Fix(number)` For more information, refer to Fix Function.
FormatCurrency	Returns an expression formatted as a currency value using the currency symbol defined in the system control panel. `FormatCurrency(Expression[,NumDigitsAfterDecimal [,IncludeLeadingDigit [,UseParensForNegativeNumbers [,GroupDigits]]]])` For more information, refer to FormatCurrency Function.
FormatDateTime	Returns an expression formatted as a date or time. `FormatDateTime(Date[, NamedFormat])` For more information, refer to FormatDateTime Function.
FormatNumber	Returns an expression formatted as a number. `FormatNumber(Expression [,NumDigitsAfterDecimal [,IncludeLeadingDigit [,UseParensForNegativeNumbers [,GroupDigits]]]])` For more information, refer to FormatNumber Function.
Hex	Returns a string representing the hexadecimal value of a number. `Hex(number)` For more information, refer to Hex Function.
Hour	Returns a whole number between 0 and 23, inclusive, representing the hour of the day.

	`Hour(time)` For more information, refer to Hour Function.
InStr	Returns the position of the first occurrence of one string within another. `InStr([start, ]string1, string2[, compare])` For more information, refer to InStr Function.
Int	Returns the integer portion of a number. `Int(number)` For more information, refer to Int Function.
IsArray	Returns a Boolean value indicating whether a variable is an array. `IsArray(varname)` For more information, refer to IsArray Function.
IsDate	Returns a Boolean value indicating whether an expression can be converted to a date. `IsDate(expression)` For more information, refer to IsDate Function.
IsEmpty	Returns a Boolean value indicating whether a variable has been initialized. `IsEmpty(expression)` For more information, refer to IsEmpty Function.
IsNull	Returns a Boolean value that indicates whether an expression contains no valid data (Null).

	`IsNull(expression)` For more information, refer to IsNull Function.
IsNumeric	Returns a Boolean value indicating whether an expression can be evaluated as a number. `IsNumeric(expression)` For more information, refer to IsNumeric Function.
LBound	Returns the smallest available subscript for the indicated dimension of an array. `LBound(arrayname[, dimension])` For more information, refer to LBound Function.
LCase	Returns a string that has been converted to lowercase. `LCase(string)` For more information, refer to LCase Function.
Left	Returns a specified number of characters from the left side of a string. `Left(string, length)` For more information, refer to Left Function.
Len	Returns the number of characters in a string or the number of bytes required to store a variable. `Len(string | varname)` For more information, refer to Len Function.

Log	Returns the natural logarithm of a number. `Log(number)` For more information, refer to [Log Function]().
LTrim	Returns a copy of a string without leading spaces (LTrim) `LTrim(string)` For more information, refer to [LTrim Function]().
Mid	Returns a specified number of characters from a string. `Mid(string, start[, length])` For more information, refer to [Mid Function]().
Minute	Returns a whole number between 0 and 59, inclusive, representing the minute of the hour. `Minute(time)` For more information, refer to [Minute Function (VBScript)]().
Month	Returns a whole number between 1 and 12, inclusive, representing the month of the year. `Month(date)` For more information, refer to [Month Function]().
MonthName	Returns a string indicating the specified month. `MonthName(month[, abbreviate])`

	For more information, refer to MonthName Function.
Replace	Returns a string in which a specified substring has been replaced with another substring a specified number of times. `Replace(expression, find, replacewith[, start[, count[, compare]]])` For more information, refer to Replace Function.
Right	Returns a specified number of characters from the right side of a string. `Right(string, length)` For more information, refer to Right Function.
Rnd	Returns a random number. `Rnd[(number)]` For more information, refer to Rnd Function.
RTrim	Returns a copy of a string without trailing spaces (RTrim). `RTrim(string)` For more information, refer to RTrim Function.
Second	Returns a whole number between 0 and 59, inclusive, representing the second of the minute. `Second(time)` For more information, refer to Second Function.
Sgn	Returns an integer indicating the sign of a number.

	`Sgn(number)` For more information, refer to Sgn Function.
Space	Returns a string consisting of the specified number of spaces. `Space(number)` For more information, refer to Space Function.
StrComp	Returns a value indicating the result of a string comparison. `StrComp(string1, string2[, compare])` For more information, refer to StrComp Function.
String	Returns a repeating character string of the length specified. `String(number, character)` For more information, refer to String Function.
TimeSerial	Returns a Variant of subtype Date containing the time for a specific hour, minute, and second. `TimeSerial(hour, minute, second)` For more information, refer to TimeSerial Function.
TimeValue	Returns a Variant of subtype Date containing the time. `TimeValue(time)` For more information, refer to TimeValue Function.
UBound	Returns the largest available subscript for the indicated dimension of an array.

	`UBound(arrayname[, dimension])` For more information, refer to UBound Function.
UCase	Returns a string that has been converted to uppercase. `UCase(string)` For more information, refer to UCase Function.
Weekday	Returns a whole number representing the day of the week. `Weekday(date, [firstdayofweek])` For more information, refer to Weekday Function.
WeekdayName	Returns a string indicating the specified day of the week. `WeekdayName(weekday, abbreviate, firstdayofweek)` For more information, refer to WeekdayName Function.
Year	Returns a whole number representing the year. `Year(date)` For more information, refer to Year Function.