FormCalc User Reference

Adobe LiveCycle Designer

Version 7.0

Adobe® LiveCycle™ Designer 7.0 FormCalc User Reference for Microsoft® Windows®
December 2004

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA.

Contents

Preface

Adobe® LiveCycle™ Designer 7.0 provides a set of tools that enables a form developer to build intelligent business documents. The form developer can incorporate calculations and scripting to create a richer experience for the recipient of the form. For example, you might use simple calculations to automatically update costs on a purchase order, or you might use more advanced scripting to modify the appearance of your form in response to the locale of the user.

To facilitate the creation of calculations, Designer provides users with FormCalc. FormCalc is a simple calculation language created by Adobe, and is modeled on common spreadsheet applications. FormCalc is simple and accessible for those with little or no scripting experience. It also follows many rules and conventions common to other scripting languages, so experienced form developers will find their skills relevant to using FormCalc.

What's in this guide?

This guide is intended for form developers using Designer who wish to incorporate FormCalc calculations in their forms. The guide provides a reference to the FormCalc functions, which are organized into chapters according to function category. The guide also provides an introduction to the FormCalc language and the building blocks that make up FormCalc expressions.

Who should read this guide?

This guide provides information to assist form developers interested in using the FormCalc language to create calculations that enhance form designs created in Designer.

Related documentation

For additional information on using FormCalc calculations in your forms, see *Creating Calculations and Scripts* in the Designer Help.

If you require more technical information about FormCalc, refer to the *Adobe XML Forms 2.1 Specification*, available from the Adobe Solutions Network (ASN) web site.

1 Introducing FormCalc

FormCalc is a simple yet powerful calculation language modeled on common spreadsheet software. Its purpose is to facilitate fast and efficient form design without requiring a knowledge of traditional scripting techniques or languages. Users new to FormCalc can expect, with the use of a few built-in functions, to create forms quickly that save end users from performing time-consuming calculations, validations, and other verifications. In this way, a form developer is able to create a basic intelligence around a form at design time that allows the resulting interactive form to react according to the data it encounters.

The built-in functions that make up FormCalc cover a wide range of areas including mathematics, dates and times, strings, finance, logic, and the Web. These areas represent the types of data that typically occur in forms, and the functions provide quick and easy manipulation of the data in a useful way.

About scripting in Designer

Within Designer, FormCalc is the default scripting language in all scripting locations, with JavaScript as the alternative. Scripting takes place on the various events that accompany each form object, and you can use a mixture of FormCalc and JavaScript on interactive, static, and dynamic forms. However, if you are using a server-based process, such as Form Server, to create forms for viewing in an internet browser, FormCalc scripts on certain form object events do not render onto the HTML form. This functionality is to prevent Internet browser errors from occurring when users work with the completed form.

2 Language Reference

Building blocks

The FormCalc language consists of a number of building blocks that make up FormCalc expressions. Each FormCalc expression is a sequence of some combination of these building blocks.

Literals

Literals are constant values that form the basis of all values that pass to FormCalc for processing. The two general types of literals are numbers and strings.

Number literals

A number literal is a sequence of mostly digits consisting of one or more of the following characters: an integer, a decimal point, a fractional segment, an exponent indicator ("e" or "E"), and an optionally signed exponent value. These are all examples of literal numbers:

- -12
- 1.5362
- 0.875
- 5.56e-2
- 1.234E10

It is possible to omit either the integer or fractional segment of a literal number, but not both. In addition, within the fractional segment, you can omit either the decimal point or the exponent value, but not both.

All number literals are internally converted to Institute of Electrical and Electronics Engineers (IEEE) 64-bit binary values. However, IEEE values can only represent a finite quantity of numbers, so certain values do not have a representation as a binary fraction. This is similar to the fact that certain values, such as 1/3, do not have a precise representation as a decimal fraction (the decimal value would need an infinite number of decimal places to be entirely accurate).

The values that do not have a binary fraction equivalent are generally number literals with more than 16 significant digits prior to their exponent. FormCalc rounds these values to the nearest representable IEEE 64-bit value in accordance with the IEEE standard. For example, the value:

```
123456789.012345678
```

rounds to the (nearest) value:

```
123456789.01234567
```

However, in a second example, the number literal:

```
99999999999999999
```

rounds to the (nearest) value:

```
100000000000000000
```

This behavior can sometimes lead to surprising results. FormCalc provides a function, Round, which returns a given number rounded to a given number of decimal places. When the given number is exactly halfway between two representable numbers, it is rounded away from zero. That is, the number is rounded up if positive and down if negative. In the following example:

```
Round(0.124, 2)
```

returns `0.12`,

and

```
Round(.125, 2)
```

returns `0.13`.

Given this convention, one might expect that:

```
Round(0.045, 2)
```

returns `0.05`.

However, the IEEE 754 standard dictates that the number literal `0.045` be approximated to `0.0449999999999999`. This approximation is closer to `0.04` than to `0.05`. Therefore,

```
Round(0.045, 2)
```

returns `0.04`.

This also conforms to the IEEE 754 standard.

IEEE 64-bit values support representations like NaN (not a number), +Inf (positive infinity), and -Inf (negative infinity). FormCalc does not support these, and expressions that evaluate to NaN, +Inf, or -Inf result in an error exception, which passes to the remainder of the expression.

String literals

A string literal is a sequence of any Unicode characters within a set of quotation marks. For example:

```
"The cat jumped over the fence."
"Number 15, Main street, California, U.S.A"
```

The string literal `""` defines an empty sequence of text characters called the empty string.

To embed a quotation mark character within a literal string, you must use two quotation marks instead of one for both the leading and trailing quotation marks. For example:

```
"The message reads: ""Warning: Insufficient Memory"""
```

All Unicode characters have an equivalent 6 character escape sequence consisting of \u followed by four hexadecimal digits. Within any literal string, it is possible to express any character, including control characters, using their equivalent Unicode escape sequence. For example:

```
"\u0047\u006f\u0066\u0069\u0073\u0068\u0021"
"\u000d" (carriage return)
"\u000a" (newline character)
```

Operators

FormCalc includes a number of operators: unary, multiplicative, additive, relational, equality, logical, and the assignment operator.

Several of the FormCalc operators have an equivalent mnemonic operator keyword. These keyword operators are useful whenever FormCalc expressions are embedded in HTML and XML source text, where the symbols less than (<), greater than (>), and ampersand (&) have predefined meanings and must be escaped. The following table lists all FormCalc operators, illustrating both the symbolic and mnemonic forms where appropriate.

Operator type	Representations
Addition	+
Division	/
Equality	== eq <> ne
Logical AND	& and
Logical OR	| or
Multiplication	*
Relational	< lt (less than) > gt (greater than) <= le (less than or equal to) >= ge (greater than or equal to)
Subtraction	-
Unary	- + not

Comments

Comments are sections of code that FormCalc does not execute. Typically comments contain information or instructions that explain the use of a particular fragment of code. FormCalc ignores all information stored in comments at run time.

You can specify a comment by using either a semi-colon (;) or a pair of slashes (//). In FormCalc, a comment extends from its beginning to the next line terminator.

Character name	Representations
Comment	; //

For example:

```
// This is a type of comment
First_Name="Tony"
Initial="C" ;This is another type of comment
Last_Name="Blue"
```

Keywords

Keywords in FormCalc are reserved words and are case-insensitive. Keywords are used as accessors, parts of expressions, special number literals, and operators.

The following table lists the FormCalc keywords. Do not use any of these words when naming objects on your form design.

and	endif	in	step
break	endwhile	infinity	then
continue	eq	le	this
do	exit	lt	throw
downto	for	nan	upto
else	foreach	ne	var
elseif	func	not	while
end	ge	null	
endfor	gt	or	
endfunc	if	return	

Identifiers

An identifier is a sequence of characters of unlimited length that denotes either a function or a method name. An identifier always begins with one of the following characters:

- Any alphabetic character (based on the Unicode letter classifications)
- Underscore (_)
- Dollar sign ($)
- Exclamation mark (!)

FormCalc identifiers are case-sensitive. That is, identifiers whose characters only differ in case are considered distinct.

Character name	Representations
Identifier	A..Z,a..z $! _

These are examples of valid identifiers:

```
GetAddr
$primary
_item
!dbresult
```

Line terminators

Line terminators are used for separating lines and improving readability.

The following table lists the valid FormCalc line terminators:

Character name	Unicode characters
Carriage Return	#xD U+000D
Line Feed	#xA
 &#D;

White space

White space characters separate various objects and mathematical operations from each other. These characters are strictly for improving readability and are irrelevant during FormCalc processing.

Character name	Unicode character
Form Feed	#xC
Horizontal Tab	#x9
Space	#x20
Vertical Tab	#xB

Expressions

Literals, operators, comments, keywords, identifiers, line terminators, and white space come together to form a list of expressions, even if the list only contains a single expression. In general, each expression in the list resolves to a value, and the value of the list as a whole is the value of the last expression in the list.

For example, consider the following scenario of two fields on a form design:

Field name	Calculations	Returns

`Field1`	`5 + Abs(Price)`
	`"Hello World"`
	`10 * 3 + 5 * 4`
		`50`

`Field2`	`10 * 3 + 5 * 4`	`50`

The value of both `Field1` and `Field2` after the evaluation of each field's expression list is `50`.

FormCalc divides the various types of expressions that make up an expression list into the following categories:

Simple

In their most basic form, FormCalc expressions are groups of operators, keywords, and literals strung together in logical ways. For example, these are all simple expressions:

```
2
"abc"
2 - 3 * 10 / 2 + 7
```

Each FormCalc expression resolves to a single value by following a traditional order of operations, even if that order is not always obvious from the expression syntax. For example, the following sets of expressions, when applied to objects on a form design, produce equivalent results:

Expression	Equivalent to	Returns
`"abc"`	`"abc"`	`abc`
`2 - 3 * 10 / 2 + 7`	`2 - (3 * (10 / 2)) + 7`	`-6`
`10 * 3 + 5 * 4`	`(10 * 3) + (5 * 4)`	`50`
`0 and 1 or 2 > 1`	`(0 and 1) or (2 >1)`	`1` (true)
`2 < 3 not 1 == 1`	`(2 < 3) not (1 == 1)`	`0` (false)

As the previous table suggests, all FormCalc operators carry a certain precedence when they appear within expressions. The following table illustrates this operator hierarchy:

Precedence	Operator
Highest	=
	(Unary) - , + , not
	* , /
	+ , -
	< , <= , > , >= , lt , le , gt , ge
	== , <> , eq , ne
	& , and
Lowest	| , or

Promoting operands

In cases where one or more of the operands within a given operation do not match the expected type for that operation, FormCalc promotes the operands to match the required type. How this promotion occurs depends on the type of operand required by the operation.

Numeric operations

When performing numeric operations involving non-numeric operands, the non-numeric operands are first promoted to their numeric equivalent. If the non-numeric operand does not successfully convert to a numeric value, its value is 0. When promoting null-valued operands to numbers, their value is always zero.

The following table provides some examples of promoting non-numeric operands:

Expression	Equivalent to	Returns
`(5 - "abc") * 3`	`(5 - 0) * 3`	`15`
`"100" / 10e1`	`100 / 10e1`	`1`
`5 + null + 3`	`5 + 0 + 3`	`8`

Boolean operations

When performing Boolean operations on non-Boolean operands, the non-Boolean operands are first promoted to their Boolean equivalent. If the non-Boolean operand does not successfully convert to a nonzero value, its value is true (1); otherwise its value is false (0). When promoting null-valued operands to a Boolean value, that value is always false (0). For example, the expression:

```
"abc" | 2
```

evaluates to `1`. That is, false | true = true, whereas

```
if ("abc") then
  10
else
  20
endif
```

evaluates to `20`.

String operations

When performing string operations on nonstring operands, the nonstring operands are first promoted to strings by using their value as a string. When promoting null-valued operands to strings, their value is always the empty string. For example, the expression:

```
concat("The total is ", 2, " dollars and ", 57, " cents.")
```

evaluates to `"The total is 2 dollars and 57 cents."`

Note: If during the evaluation of an expression an intermediate step yields NaN, +Inf, or -Inf, FormCalc generates an error exception and propagates that error for the remainder of the expression. As such, the expression's value will always be 0. For example:

```
3 / 0 + 1
```

evaluates to 0.

Assignment

An assignment expression sets the property identified by a given accessor to be the value of a simple expression. For example:

```
$template.purchase_order.name.first = "Tony"
```

This sets the value of the form design object “first” to `Tony`.

For more information on using accessors, see “Accessors” on page 20.

Logical OR

A logical OR expression returns either true (1) if at least one of its operands is true (1), or false (0) if both operands are false (0). If both operands are null, the expression returns null.

Expression	Character representation
Logical OR	| or

These are examples of using the logical OR expression:

Expression	Returns
`1 or 0`	`1` (true)
`0 | 0`	`0` (false)
`0 or 1 | 0 or 0`	`1` (true)

Logical AND

A logical AND expression returns either true (1) if both operands are true (1), or false if at least one of its operands is false (0). If both operands are null, the expression returns null.

Expression	Character representation
Logical AND	& and

These are examples of using the logical AND expression:

Expression	Returns
`1 and 0`	`0` (false)
`0 & 0`	`1` (true)
`0 and 1 & 0 and 0`	`0` (false)

Unary

A unary expression returns different results depending on which of the unary operators is used.

Expression	Character representation	Returns
Unary	-	The arithmetic negation of the operand, or null if the operand is null.
	+	The arithmetic value of the operand (unchanged), or null if its operand is null.
	not	The logical negation of the operand.

Note: The arithmetic negation of a null operand yields the result null, whereas the logical negation of a null operand yields the Boolean result true. This is justified by the common sense statement: If null means nothing, then "not nothing" should be something.

These are examples of using the unary expression:

Expression	Returns
`-(17)`	`-17`
`-(-17)`	`17`
`+(17)`	`17`
`+(-17)`	`-17`
`not("true")`	`1` (true)
`not(1)`	`0` (false)

Equality and inequality

Equality and inequality expressions return the result of an equality comparison of its operands.

Expression	Character representation	Returns
Equality	== eq	True (1) when both operands compare identically, and false (0) if they do not compare identically.
Inequality	<> ne	True (1) when both operands do not compare identically, and false (0) if they compare identically.

The following special cases also apply when using equality operators:

- If either operand is null, a null comparison is performed. Null-valued operands compare identically whenever both operands are null, and compare differently whenever one operand is not null.
- If both operands are references, both operands compare identically when they both refer to the same object, and compare differently when they do not refer to the same object.
- If both operands are string valued, a locale-sensitive lexicographic string comparison is performed on the operands. Otherwise, if they are not both null, the operands are promoted to numeric values, and a numeric comparison is performed.

These are examples of using the equality and inequality expressions:

Expression	Returns
`3 == 3`	`1` (true)
`3 <> 4`	`1` (true)
`"abc" eq "def"`	`0` (false)
`"def" ne "abc"`	`1` (true)
`5 + 5 == 10`	`1` (true)
`5 + 5 <> "10"`	`0` (false)

Relational

A relational expression returns the Boolean result of a relational comparison of its operands.

Expression	Character representation	Returns
Relational	< lt	True (1) when the first operand is less than the second operand, and false (0) when the first operand is larger than the second operand.
	> gt	True (1) when the first operand is greater than the second operand, and false (0) when the first operand is less than the second operand.
	<= le	True (1) when the first operand is less than or equal to the second operand, and false (0) when the first operand is greater than the second operand.
	>= ge	True (1) when the first operand is greater than or equal to the second operand, and false (0) when the first operand is less than the second operand.

The following special cases also apply when using relational operators:

- If either operand is null valued, a null comparison is performed. Null-valued operands compare identically whenever both operands are null and the relational operator is less-than-or-equal or greater than or equal, and compare differently otherwise.
- If both operands are string valued, a locale-sensitive lexicographic string comparison is performed on the operands. Otherwise, if they are not both null, the operands are promoted to numeric values, and a numeric comparison is performed.

These are examples of using the relational expression:

Expression	Returns
`3 < 3`	`0` (false)
`3 > 4`	`0` (false)
`"abc" <= "def"`	`1` (true)
`"def" > "abc"`	`1` (true)
`12 >= 12`	`1` (true)
`"true" < "false"`	`0` (false)

If expressions

An if expression is a conditional statement that evaluates a given simple expression for truth, and then returns the result of a list of expressions that correspond to the truth value. If the initial simple expression evaluates to false (0), FormCalc examines any elseif and else conditions for truth and returns the results of their expression lists if appropriate.

Expression	Syntax	Returns

If	`if ( simple expression ) then`
	`  list of expressions`
	`elseif ( simple expression ) then`
	`  list of expressions`
	`else`
	`  list of expressions`
	`endif`
		The result of the list of expressions associated with any valid conditions stated in the if expression.
		Note: You are not required to have any elseif(...) or else statements as part of your if expression, but you must state the end of the expression with endif.

These are examples of using the if expression:

Expression	Returns

`if ( 1 < 2 ) then`
`  1`
`endif`
	`1`

`if ( "abc" > "def") then`
`  1 and 0`
`else`
`  0`
`endif`
	`0`

`if ( Field1 < Field2 ) then`
`  Field3 = 0`
`elseif ( Field1 > Field2 ) then`
`  Field3 = 40`
`elseif ( Field1 = Field2 ) then`
`  Field3 = 10`
`endif`
	Varies with the values of `Field1` and `Field2`. For example, if Field1 is 20 and Field2 is 10, then this expression sets Field3 to 40.

Variables

Within your calculations, FormCalc allows you to create and manipulate variables for storing data. The name you assign to each variable you create must be a unique identifier.

For example, the following FormCalc expressions define the userName variable and set the value of a text field to be the value of userName.

```
var userName = "Tony Blue"
TextField1.rawValue = userName
```

You can reference variables that you define in the Variables tab of the Form Properties dialog box in the same way. The following FormCalc expression uses the Concat function to set the value of the text field using the form variables salutation and name.

```
TextField1.rawValue = Concat(salutation, name)
```

Note: A variable you create using FormCalc will supersede a similarly named variable you define in the Variables tab of the Form Properties dialog box.

Accessors

FormCalc provides access to form design object properties and values. An accessor is a mechanism that enables you to assign or retrieve specific object values and properties.

The following example demonstrates both assigning and retrieving object values:

```
Invoice.VAT = Invoice.Total * (8 / 100)
```

In this case the accessor `Invoice.VAT` is assigned the value of `Invoice.Total * (8 / 100)`.

In the context of form design, a qualified hierarchy enables access to all the objects on the form design. Accessors provide predefined syntax that makes navigation of this hierarchy easier.

The following table outlines the correct syntax for all accessors within FormCalc.

Notation	Description

$ (FormCalc)
this (JavaScript)
	Refers to the current field or object. For example:
	`$ = "Tony Blue"`
	`this.rawValue = "Tony Blue"`
	The above examples set the value of the current field or object to `Tony Blue` using both FormCalc and JavaScript.
	Note: These accessors must appear at the beginning of a hierarchy reference, that is, before the first period.

!	Represents the root of the data model, xfa.datasets. For example:
	`!dbresults`
	is equivalent to:
	`xfa.datasets.dbresults`
	Note: This accessor must appear at the beginning of a hierarchy reference, that is, before the first period.

$data (FormCalc)
xfa.datasets.data (JavaScript)
	Represents the root of the data model, xfa.datasets.data. For example:
	`$data.purchaseOrder.total`
	is equivalent to:
	`xfa.datasets.data.purchaseOrder.total`
	Note: This accessor must appear at the beginning of a hierarchy reference, that is, before the first period.

Notation	Description

$template (FormCalc)
xfa.template (JavaScript)
	Represents the root of the template model, xfa.template. For example:
	`$template.purchaseOrder.item[1]`
	is equivalent to:
	`xfa.template.purchaseOrder.item[1]`
	Note: This accessor must appear at the beginning of a hierarchy reference, that is, before the first period.

$form (FormCalc)
xfa.form (JavaScript)
	Represents the root of the form model, xfa.form. For example:
	`$form.purchaseOrder.tax[0]`
	is equivalent to stating:
	`xfa.form.purchaseOrder.tax[0]`
	Note: This accessor must appear at the beginning of a hierarchy reference, that is, before the first period.

$layout (FormCalc)
xfa.layout (JavaScript)
	Represents the root of the layout model, xfa.layout. For example:
	`$layout.purchaseOrder.tax[0]`
	is equivalent to stating:
	`xfa.layout.purchaseOrder.tax[0]`

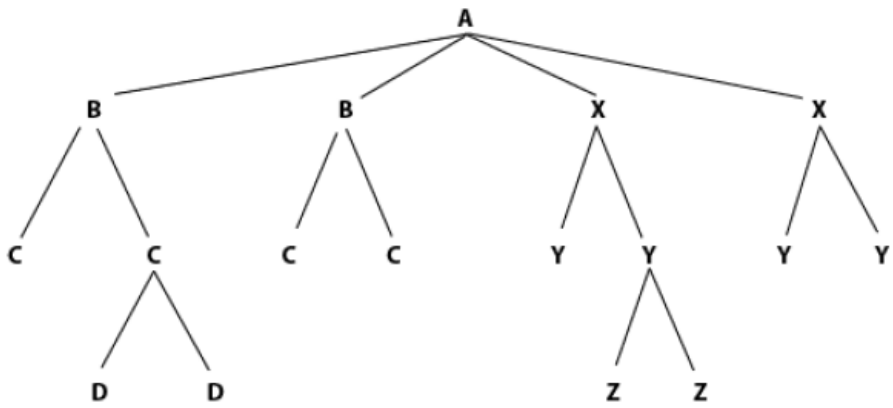
$record (FormCalc) xfa.record (JavaScript)	Represents the current record of a collection of data, such as from an XML file. For example: `$record.header.txtOrderedByCity` references the `txtOrderedByCity` node within the `header` node of the current XML data. **Note:** This accessor must appear at the beginning of a hierarchy reference, that is, before the first period.

$event (FormCalc)
xfa.event (JavaScript)
	Represents the current form object event. For example:
	`$event.name`
	is equivalent to:
	`xfa.event.name`
	Note: This accessor must appear at the beginning of a hierarchy reference, that is, before the first period.

$host (FormCalc)
xfa.host (JavaScript)
	Represents the host object. For example:
	`$host.name`
	is equivalent to:
	`xfa.host.name`
	Note: This accessor must appear at the beginning of a hierarchy reference, that is, before the first period.

Notation	Description

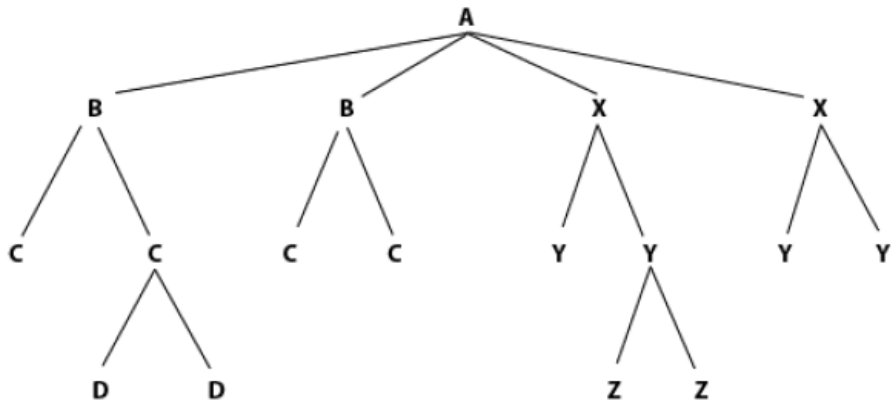
*	Selects all form objects within a given container regardless of name, or selects all objects with a similar name. This accessor cannot begin a SOM expression.
	For example, the following expression selects all containers on a form design:
	`$template.*`
	The following expression selects all objects named `item`:
	`form1.item[*]`

..	You can use two dots at any point in your SOM expression to search for objects that are a part of any subcontainer of the current container. For example, the expression `A..B` means locate the node `A` (as usual), and find a descendant of `A` called `B`.
	 
	Using the example tree above:
	`A..C`
	is equivalent to:
	`A.B[0].C[0]`
	because `C[0]` is the first `C` node FormCalc encounters on its search. As a second example:
	`A..C[*]`
	returns the two `C` nodes on the extreme left, because in this example, `A..C[*]` is equivalent to `A.B[0].C[*]`.

#	Matches an unnamed form design object or property. This accessor is useful if both a property and an object have the same name. The number sign (#) ensures that the script accesses the property value. For example: `purchaseOrder.#name` This expression returns the actual name of the purchase order object, in this case `purchaseOrder`.

Notation	Description

[]	An array referencing syntax. FormCalc treats the collection of accessible objects with the same name as an array. Note that all array references are zero-based.
	In order to construct an array element reference, place square brackets ([ ]) after a qualified accessor name, and enclose within the brackets one of the following expressions:
	• `[ n ]`
	Where `n` is an absolute occurrence index number beginning at 0. An occurrence number that is out of range is an error. Occurrence numbers in SOM syntax are not expressions.
	Only numbers are valid. For example:
	`$xfa.template.Quantity[3]`
	refers to the fourth occurrence of Quantity.
	• `[ +/- n ]`
	Where `n` indicates an occurrence relative to the occurrence of the object making the reference. Positive values yield higher occurrence numbers while negative values yield lower occurrence numbers. For example:
	`$xfa.template.Quantity[+2]`
	This expression yields the occurrence of `Quantity` whose occurrence number is two more than the occurrence number of the container making the reference. For example, if this reference was attached to `Amount[2]`, the reference would be the same as:
	`$xfa.template.Quantity[4]`
	If the computed index number is out of range, it is an error.
	The most common use of this syntax is for locating the previous or next occurrence of a particular field. For example, every occurrence of the field `Amount` (except the first) might use `Amount[-1]` to get the value of the previous amount field.
	• `[*]`
	Indicates multiple occurrences of the object. The first named object is found, and objects of the same name that are siblings to the first are returned. Note that using this notation returns in a collection of objects. For example:
	`Quantity[*]`
	This expression refers to all objects with a name of `Quantity` that are siblings to the first `Quantity` found.

Notation	Description
[ ] (Continued)	 Using the tree for reference, these expressions return the following values: • `A.B[*]` Both B nodes • `A.B.C[*]` Two `C` nodes on the extreme left. `A.B` resolves to the first `B` node on the left, and the `C[*]` is evaluated relative to that node. • `A.B[*].C` The first and the third `C` nodes from the left. `A.B[*]` resolves to both `B` nodes, and the `C` is evaluated relative to both of those `B` nodes. • `A.B[*].C[1]` The second and fourth `C` nodes from the left. `A.B[*]` resolves to both `B` nodes, and the `C[1]` is evaluated relative to both of those `B` nodes. • `A.B[*].C[*]` All four `C` nodes. • `A.*` Both `B` nodes and both `X` nodes • `A.X.*` Two left-most `Y` nodes.

Method calls

Designer defines a variety of methods for all objects on a form design. FormCalc provides access to these methods and allows you to use them to act upon objects and their properties. Similar to a function call, you invoke methods by passing arguments to them in a specific order. The number and type of arguments in each method are specific to each object type.

Note: Different form design objects support different methods.

Function calls

FormCalc supports a large set of built-in functions with a wide range of capabilities. The names of the functions are case-insensitive, but unlike keywords, FormCalc does not reserve the names of the functions. This means that calculations on forms with objects whose names coincide with the names of FormCalc functions do not conflict.

Functions may or may not require some set of arguments to execute and return a value. Many functions have arguments that are optional, meaning it is up to you to decide if the argument is necessary for the particular situation.

FormCalc evaluates all function arguments in order, beginning with the lead argument. If an attempt is made to pass less than the required number of arguments to a function, the function generates an error exception.

Each function expects each argument in a particular format, either as a number literal or string literal. If the value of an argument does not match what a function expects, FormCalc converts the value. For example:

```
Len(35)
```

The Len function actually expects a literal string. In this case, FormCalc converts the argument from the number `35` to the string `"35"`, and the function evaluates to `2`.

However, in the case of a string literal to number literal, the conversion is not so simple. For example:

```
Abs("abc")
```

The Abs function expects a number literal. FormCalc converts the value of all string literals as 0. This can cause problems in functions where a `0` value forces an error, such as in the case of the Apr function.

Some function arguments only require integral values; in such cases, the passed arguments are always promoted to integers by truncating the fractional part.

Here is a summary of the key properties of built-in functions:

- Built-in function names are case-insensitive.
- The built-in functions are predefined, but their names are not reserved words. This means that the built-in function Max never conflicts with an object, object property, or object method named Max.
- Many of the built-in functions have a mandatory number of arguments, which can be followed by a optional number of arguments.
- A few built-in functions, Avg, Count, Max, Min, Sum, and Concat, accept an indefinite number of arguments.

For a complete listing of all the FormCalc functions, see the "Alphabetical Functions List" on page 26.

3 Alphabetical Functions List

The following table lists all available FormCalc functions, provides a description of each function, and identifies the category type to which each function belongs.

Function	Description	Type
"Abs" on page 31	Returns the absolute value of a numeric value or expression.	Arithmetic
"Apr" on page 62	Returns the annual percentage rate for a loan.	Financial
"At" on page 86	Locates the starting character position of a string within another string.	String
"Avg" on page 32	Evaluates a set of number values and/or expressions and returns the average of the non-null elements contained within that set.	Arithmetic
"Ceil" on page 33	Returns the whole number greater than or equal to a given number.	Arithmetic
"Choose" on page 74	Selects a value from a given set of parameters.	Logical
"Concat" on page 87	Returns the concatenation of two or more strings.	String
"Count" on page 34	Evaluates a set of values and/or expressions and returns the number of non-null elements contained within the set.	Arithmetic
"CTerm" on page 63	Returns the number of periods needed for an investment earning a fixed, but compounded, interest rate to grow to a future value.	Financial
"Date" on page 48	Returns the current system date as the number of days since the epoch.	Date and Time
"Date2Num" on page 49	Returns the number of days since the epoch, given a date string.	Date and Time
"DateFmt" on page 50	Returns a date format string, given a date format style.	Date and Time
"Decode" on page 88	Returns the decoded version of a given string.	String
"Encode" on page 89	Returns the encoded version of a given string.	String
"Eval" on page 80	Returns the value of a given form calculation.	Miscellaneous
"Exists" on page 75	Determines whether the given parameter is an accessor to an existing object.	Logical
"Floor" on page 35	Returns the largest whole number that is less than or equal to the given value.	Arithmetic

Function	Description	Type
"Format" on page 90	Formats the given data according to the specified picture format string.	String
"FV" on page 64	Returns the future value of consistent payment amounts made at regular intervals at a constant interest rate.	Financial
"Get" on page 107	Downloads the contents of the given URL.	URL
"HasValue" on page 76	Determines whether the given parameter is an accessor with a non-null, non-empty, or non-blank value.	Logical
"IPmt" on page 65	Returns the amount of interest paid on a loan over a set period of time.	Financial
"IsoDate2Num" on page 51	Returns the number of days since the epoch, given an valid date string.	Date and Time
"IsoTime2Num" on page 52	Returns the number of milliseconds since the epoch, given a valid time string.	Date and Time
"Left" on page 91	Extracts a specified number of characters from a string, starting with the first character on the left.	String
"Len" on page 92	Returns the number of characters in a given string.	String
"LocalDateFmt" on page 53	Returns a localized date format string, given a date format style.	Date and Time
"LocalTimeFmt" on page 54	Returns a localized time format string, given a time format style.	Date and Time
"Lower" on page 93	Converts all uppercase characters within a specified string to lowercase characters.	String
"Ltrim" on page 94	Returns a string with all leading white space characters removed.	String
"Max" on page 36	Returns the maximum value of the non-null elements in the given set of numbers.	Arithmetic
"Min" on page 37	Returns the minimum value of the non-null elements of the given set of numbers.	Arithmetic
"Mod" on page 38	Returns the modulus of one number divided by another.	Arithmetic
"NPV" on page 66	Returns the net present value of an investment based on a discount rate and a series of periodic future cash flows.	Financial
"Null" on page 81	Returns the null value. The null value means no value.	Miscellaneous
"Num2Date" on page 55	Returns a date string, given a number of days since the epoch.	Date and Time
"Num2GMTime" on page 56	Returns a GMT time string, given a number of milliseconds from the epoch.	Date and Time

Function	Description	Type
"Num2Time" on page 57	Returns a time string, given a number of milliseconds from the epoch.	Date and Time
"Oneof" on page 77	Returns true (1) if a value is in a given set, and false (0) if it is not.	Logical
"Parse" on page 95	Analyzes the given data according to the given picture format.	String
"Pmt" on page 67	Returns the payment for a loan based on constant payments and a constant interest rate.	Financial
"Post" on page 108	Posts the given data to the specified URL.	URL
"PPmt" on page 68	Returns the amount of principal paid on a loan over a period of time.	Financial
"Put" on page 110	Uploads the given data to the specified URL.	URL
"PV" on page 69	Returns the present value of an investment of periodic constant payments at a constant interest rate.	Financial
"Rate" on page 70	Returns the compound interest rate per period required for an investment to grow from present to future value in a given period.	Financial
"Ref" on page 82	Returns a reference to an existing object.	Miscellaneous
"Replace" on page 96	Replaces all occurrences of one string with another within a specified string.	String
"Right" on page 97	Extracts a number of characters from a given string, beginning with the last character on the right.	String
"Round" on page 39	Evaluates a given numeric value or expression and returns a number rounded to the given number of decimal places.	Arithmetic
"Rtrim" on page 98	Returns a string with all trailing white space characters removed.	String
"Space" on page 99	Returns a string consisting of a given number of blank spaces.	String
"Str" on page 100	Converts a number to a character string. FormCalc formats the result to the specified width and rounds to the specified number of decimal places.	String
"Stuff" on page 101	Inserts a string into another string.	String
"Substr" on page 102	Extracts a portion of a given string.	String
"Sum" on page 40	Returns the sum of the non-null elements of a given set of numbers.	Arithmetic

Function	Description	Type
"Term" on page 71	Returns the number of periods needed to reach a given future value from periodic constant payments into an interest-bearing account.	Financial
"Time" on page 58	Returns the current system time as the number of milliseconds since the epoch.	Date and Time
"Time2Num" on page 59	Returns the number of milliseconds since the epoch, given a time string.	Date and Time
"TimeFmt" on page 60	Returns a time format, given a time format style.	Date and Time
"UnitType" on page 83	Returns the units of a unitspan. A unitspan is a string consisting of a number followed by a unit name.	Miscellaneous
"UnitValue" on page 84	Returns the numeric value of a measurement with its associated unitspan, after an optional unit conversion.	Miscellaneous
"Upper" on page 104	Converts all lowercase characters within a string to uppercase.	String
"Uuid" on page 103	Returns a Universally Unique Identifier (UUID) string to use as an identification method.	String
"Within" on page 78	Returns true (1) if the test value is within a given range, and false (0) if it is not.	Logical
"WordNum" on page 105	Returns the English text equivalent of a given number.	String

4 Arithmetic Functions

These functions perform a range of mathematical operations.

Functions

Abs

Returns the absolute value of a numeric value or expression, or returns null if the value or expression is null.

Syntax

`Abs(n1)`

Parameters

Parameter	Description
n1	A numeric value or expression to evaluate.

Note: FormCalc follows the IEEE-754 international standard when handling floating point numeric values. For more information, see "Number literals" on page 8.

Examples

The following expressions are examples of using the Abs function:

Expression	Returns
`Abs(1.03)`	`1.03`
`Abs(-1.03)`	`1.03`
`Abs(0)`	`0`

Avg

Evaluates a set of number values and/or expressions and returns the average of the non-null elements contained within that set.

Syntax

```
Avg(n1 [, n2 ...])
```

Parameters

Parameter	Description
n1	The first numeric value or expression of the set.
n2 (*optional*)	Additional numeric values or expressions.

Note: FormCalc follows the IEEE-754 international standard when handling floating point numeric values. For more information, see "Number literals" on page 8.

Examples

The following expressions are examples of using the Avg function:

Expression	Returns
`Avg(0, 32, 16)`	`16`
`Avg(2.5, 17, null)`	`9.75`
`Avg(Price[0], Price[1], Price[2], Price[3])`	The average value of the first four non-null occurrences of `Price`.
`Avg(Quantity[*])`	The average value of all non-null occurrences of `Quantity`.

Ceil

Returns the whole number greater than or equal to a given number, or returns null if its parameter is null.

Syntax

`Ceil(n)`

Parameters

Parameter	Description
n	Any numeric value or expression. The function returns `0` if *n* is not a numeric value or expression.

Note: FormCalc follows the IEEE-754 international standard when handling floating point numeric values. For more information, see "Number literals" on page 8.

Examples

The following expressions are examples of using the Ceil function:

Expression	Returns
`Ceil(2.5875)`	`3`
`Ceil(-5.9)`	`-5`
`Ceil("abc")`	`0`
`Ceil(A)`	100 if the value of `A` is 99.999

Count

Evaluates a set of values and/or expressions and returns the count of non-null elements contained within the given set.

Syntax

```
Count(n1 [, n2 ...])
```

Parameters

Parameter	Description
n1	A numeric value or expression.
n2 (optional)	Additional numeric values and/or expressions.

Note: FormCalc follows the IEEE-754 international standard when handling floating point numeric values. For more information, see "Number literals" on page 8.

Examples

The following expressions are examples of using the Count function:

Expression	Returns
`Count("Tony", "Blue", 41)`	`3`
`Count(Customers[*])`	The number of non-null occurrences of `Customers`.
`Count(Coverage[2], "Home", "Auto")`	`3`, provided the third occurrence of `Coverage` is non-null.

Floor

Returns the largest whole number that is less than or equal to the given value.

Syntax

`Floor(n)`

Parameters

Parameter	Description
n	Any numeric value or expression.

Note: FormCalc follows the IEEE-754 international standard when handling floating point numeric values. For more information, see "Number literals" on page 8.

Examples

The following expressions are examples of using the Floor function:

Expression	Returns
`Floor(21.3409873)`	`21`
`Floor(5.999965342)`	`5`
`Floor(3.2 * 15)`	`48`

Max

Returns the maximum value of the non-null elements in the given set of numbers.

Syntax

```
Max(n1 [, n2 ...])
```

Parameters

Parameter	Description
n1	A numeric value or expression.
n2 (*optional*)	Additional numeric values and/or expressions.

Note: FormCalc follows the IEEE-754 international standard when handling floating point numeric values. For more information, see "Number literals" on page 8.

Examples

The following expressions are examples of using the Max function:

Expression	Returns
`Max(234, 15, 107)`	`234`
`Max("abc", 15, "Tony Blue")`	`15`
`Max("abc")`	`0`
`Max(Field1[*], Field2[0])`	Evaluates the non-null occurrences of `Field1` as well as the first occurrence of `Field2`, and returns the highest value.
`Max(Min(Field1[*], Field2[0]), Field3, Field4)`	The first expression evaluates the non-null occurrences of `Field1` as well as the first occurrence of `Field2`, and returns the lowest value. The final result is the maximum of the returned value compared against the values of `Field3` and `Field4`. See also "Min" on page 37.

Min

Returns the minimum value of the non-null elements of the given set of numbers.

Syntax

```
Min(n1 [, n2 ...])
```

Parameters

Parameter	Description
n1	A numeric value or expression.
n2 (*optional*)	Additional numeric values and/or expressions.

Note: FormCalc follows the IEEE-754 international standard when handling floating point numeric values. For more information, see "Number literals" on page 8.

Examples

The following expressions are examples of using the Min function:

Expression	Returns
`Min(234, 15, 107)`	`15`
`Min("abc", 15, "Tony Blue")`	`15`
`Min("abc")`	`0`
`Min(Field1[*], Field2[0])`	Evaluates the non-null occurrences of `Sales_July` as well as the first occurrence of `Sales_August`, and returns the lowest value.
`Min(Max(Field1[*], Field2[0]), Field3, Field4)`	The first expression evaluates the non-null occurrences of `Field1` as well as the first occurrence of `Field2`, and returns the highest value. The final result is the minimum of the returned value compared against the values of `Field3` and `Field4`. See also "Max" on page 36.

Mod

Returns the modulus of one number divided by another. The modulus is the remainder of the division of the dividend by the divisor. The sign of the remainder always equals the sign of the dividend.

Syntax

`Mod(n1, n2)`

Parameters

Parameter	Description
n1	The dividend, a numeric value or expression.
n2	The divisor, a numeric value or expression.

If *n1* and/or *n2* are not numeric values or expressions, the function returns 0.

Note: FormCalc follows the IEEE-754 international standard when handling floating point numeric values. For more information, see "Number literals" on page 8.

Examples

The following expressions are examples of using the Mod function:

Expression	Returns
`Mod(64, -3)`	`1`
`Mod(-13,3)`	`-1`
`Mod("abc", 2)`	`0`
`Mod(X[0], Y[9])`	The first occurrence of `X` is used as the dividend and the tenth occurrence of `Y` is used as the divisor.
`Mod(Round(Value[4], 2), Max(Value[*]))`	The first fifth occurrence of `Value` rounded to two decimal places is used as the dividend and the highest of all non-null occurrences of `Value` is used as the divisor. See also "Max" on page 36 and "Round" on page 39.

Round

Evaluates a given numeric value or expression and returns a number rounded to a given number of decimal places.

Syntax

```
Round(n1 [, n2])
```

Parameters

Parameter	Description
n1	A numeric value or expression to be evaluated.
n2 (*optional*)	The number of decimal places with which to evaluate *n1* to a maximum of 12. If you do not include a value for *n2*, or if *n2* is invalid, the function assumes the number of decimal places is 0.

Note: FormCalc follows the IEEE-754 international standard when handling floating point numeric values. For more information, see "Number literals" on page 8.

Examples

The following expressions are examples of using the Round function:

Expression	Returns
`Round(12.389764537, 4)`	`12.3898`
`Round(20/3, 2)`	`6.67`
`Round(8.9897, "abc")`	`9`
`Round(FV(400, 0.10/12, 30*12), 2)`	`904195.17`. This takes the value evaluated using the FV function and rounds it to two decimal places. See also "FV" on page 64.
`Round(Total_Price, 2)`	Rounds off the value of `Total_Price` to two decimal places.

Sum

Returns the sum of the non-null elements of a given set of numbers.

Syntax

```
Sum(n1 [, n2 ...])
```

Parameters

Parameter	Description
n1	A numeric value or expression.
n2 (*optional*)	Additional numeric values and/or expressions.

Note: FormCalc follows the IEEE-754 international standard when handling floating point numeric values. For more information, see "Number literals" on page 8.

Examples

The following expressions are examples of using the Sum function:

Expression	Returns
`Sum(2, 4, 6, 8)`	`20`
`Sum(-2, 4, -6, 8)`	`4`
`Sum(4, 16, "abc", 19)`	`39`
`Sum(Amount[2], Amount[5])`	Totals the third and sixth occurrences of `Amount`.
`Sum(Round(20/3, 2), Max(Amount[*]), Min(Amount[*]))`	Totals the value of 20/3 rounded to two decimal places, as well as the largest and smallest non-null occurrences of `Amount`. See also "Max" on page 36, "Min" on page 37, and "Round" on page 39.

5 Date and Time Functions

Functions in this section deal specifically with creating and manipulating date and time values.

Functions

Structuring dates and times

Locales

A locale is a standard term used when developing international standards to identify a particular nation (language, country, or both). For the purposes of FormCalc, a locale defines the format of dates and times relevant to a specific nation or region, enabling end users to use the date and time formats they are accustomed to.

Each locale is comprised of a unique string of characters called a locale identifier. The composition of these strings is controlled by the international standards organization (ISO) Internet Engineering Task Force (IETF), a working group of the Internet Society (www.isoc.org).

Locale identifiers are consist of a language part, a country part, or both. The following table lists valid locales for this release of Designer:

Identifier (ISO Code)	Locale
bg_BG	Bulgarian (Bulgaria)
zh_HK	Chinese (Hong Kong)
zh_CN	Simplified Chinese (People's Republic of China)
zh_TW	Traditional Chinese (Taiwan)
hr_HR	Croatian (Croatia)
cs_CZ	Czech (Czech Republic)
da_DK	Danish (Denmark)
nl_BE	Dutch (Belgium)
nl_NL	Dutch (Netherlands)
en-AU	English (Australia)
en_CA	English (Canada)
en_IN	English (India)
en_IE	English (Ireland)
en_NZ	English (New Zealand)
en_ZA	English (South Africa)
en_GB_EURO	English (United Kingdom Euro)
en_US	English (United States of America)
et_EE	Estonian (Estonia)
fi_FI	Finnish (Finland)
fr_BE	French (Belgium)
fr_CA	French (Canada)

Identifier (ISO Code)	Locale
afr_FR	French (France)
fr_LU	French (Luxembourg)
fr_CH	French (Switzerland)
de_AT	German (Austria)
de_DE	German (Germany)
de_LU	German (Luxembourg)
de_CH	German (Switzerland)
el_GR	Greek (Greece)
hu_HU	Hungarian (Hungary)
it_IT	Italian (Italy)
it_CH	Italian (Switzerland)
ja_JP	Japanese (Japan)
ko_KR	Korean (Republic of Korea)
lv_LV	Latvian (Latvia)
lt_LT	Lithuanian (Lithuania)
no_NO_B	Norwegian (Bokmal Norway)
no_NO_NY	Norwegian (Nynorsk Norway)
pl_PL	Polish (Poland)
pt_BR	Portuguese (Brazil)
pt_PT	Portuguese (Portugal)
ro_RO	Romanian (Romania)
ru_RU	Russian (Russia)
sh_BA	Serbo-Croatian (Bosnia and Herzegovina)
sh_HR	Serbo-Croatian (Croatia)
sh_CS	Serbo-Croatian (Republic of Serbia and Montenegro)
sk_SK	Slovak (Slovakia)
sl_SI	Slovenian (Slovenia)
es_EC	Spanish (Ecuador)
es_SV	Spanish (El Salvador)
es_GT	Spanish (Guatemala)
es_HN	Spanish (Honduras)

Identifier (ISO Code)	Locale
es_NI	Spanish (Nicaragua)
es_PA	Spanish (Panama)
es_PY	Spanish (Paraguay)
es_PR	Spanish (Puerto Rico)
es_UY	Spanish (Uruguay)
es-AR	Spanish (Argentina)
es_BO	Spanish (Bolivia)
es-CL	Spanish (Chile)
es_CO	Spanish (Columbia)
es_CR	Spanish (Costa Rica)
es_DO	Spanish (Dominican Republic)
es_MX	Spanish (Mexico)
es_PE	Spanish (Peru)
es_ES	Spanish (Spain)
es_VE	Spanish (Venezuela)
sv_SE	Swedish (Sweden)
tr_TR	Turkish (Turkey)
uk_UA	Ukrainian (Ukraine)

Usually, both elements of a locale are important. For example, the names of weekdays and months in English for Canada and in Great Britain are formatted identically, but dates are formatted differently. So, specifying an English language locale is not enough. Conversely, specifying only a country as the locale is not enough. For example, Canada has different date formats for English and French.

In general, every application operates in an environment where a locale is present. This locale is known as the ambient locale. In some circumstances, an application might operate on a system, or within an environment, where a locale is not present. In these rare cases, the ambient locale is set to a default of English United States (en_US). This locale is known as a default locale.

Epoch

Date values and time values have an associated origin or epoch, which is a moment in time from which things begin. Any date value prior to its epoch is invalid, as is any time value prior to its epoch.

The unit of value for all date functions is the number of days since the epoch. The unit of value for all time functions is the number of milliseconds since the epoch.

Designer defines day 1 for the epoch for all date functions as Jan 1, 1900, and millisecond one for the epoch for all time functions is midnight, 00:00:00, Greenwich Mean Time (GMT). This definition means that negative time values may be returned to users in time zones east of GMT.

Date formats

A date format is a shorthand specification of how a date appears. It consists of various punctuation marks and symbols representing the formatting that the date must use. The following table lists examples of date formats:

Date format	Example
MM/DD/YY	11/11/78
DD/MM/YY	25/7/85
MMMM DD, YYYY	March 10, 1964

The format of dates is governed by an ISO standard. Each country or region specifies its own date formats. There are five general categories of date formats: default, short, medium, long, and full. The following table gives some examples of different date formats from different locales for each of the five categories.

Date format category	Locale identifier	Date format	Example
Default (Typically follows the medium format definition, but not always.)	en_US	MMM D, YYYY	Feb 10, 1970
Short (Numeric)	en_GB	M/D/YY	8/18/92 8/4/92
Medium (0 padded 2 digit month value)	fr_CA	YY/MM/DD	99/06/26
Long (Full month names)	de_DE	D. MMMM YYYY	17. Juni 1989
Full (Includes the weekday name)	fr_FR	EEEE,' le 'D MMMM YYYY	Lundi, le 29 Octobre, 1990

Time formats

A time format is a shorthand specification to format a time. It consists of punctuations, literals, and pattern symbols. The following table lists examples of time formats:

Time format	Example
h:MM A	7:15 PM
HH:MM:SS	21:35:26
HH:MM:SS 'o''clock' A Z	14:20:10 o’clock PM EDT

Time formats are governed by an ISO standard. Each nation specifies the form of its default, short, medium, long, and full time formats. The locale is responsible for identifying the format of times that conform to the standards of that nation.

The following table gives some examples of different date formats from different locales for each of the five categories.

Time format category	Locale identifier	Time format	Example
Default (Typically follows the medium format definition for the locale, but not always.)	en_US	h:mm:ss a	7:45:15 a
Short (24 hour system)	en_GB	HH:mm	14:13
Medium (24 hour system and a value for seconds.)	fr_CA	HH:mm:ss	12:15:50
Long (24 hour system, value for seconds, and the hour differential between the locale and Greenwich Mean Time.)	de_DE	HH:mm:ss z	14:13:13 -04:00
Full (Includes the weekday name)	fr_FR	HH' h 'mm z	14 h 13 -0400

Date and time picture formats

The following symbols must be used to create date and time patterns for date/time fields.

Note: The comma (,), dash (-), colon (:), slash (/), period (.), and space ( ) are treated as literal values and can be included anywhere in a pattern. To include a phrase in a pattern, delimit the text string with single quotation marks ('). For example, `'Your payment is due no later than' MM-DD-YY`.

Date symbol	Description
`D`	1 or 2 digit (1-31) day of the month
`DD`	Zero-padded 2 digit (01-31) day of the month
`J`	1, 2, or 3 digit (1-366) day of the year
`JJJ`	Zero-padded, three-digit (001-366) day of the year
`M`	One- or two-digit (1-12) month of the year
`MM`	Zero-padded, two-digit (01-12) month of the year
`MMM`	Abbreviated month name
`MMMM`	Full month name
`E`	One-digit (1-7) day of the week, where (1=Sunday)
`EEE`	Abbreviated weekday name
`EEEE`	Full weekday name
`YY`	Two-digit year, where 00=2000, 29=2029, 30=1930, and 99=1999

Date symbol	Description
YYYY	Four-digit year
G	Era name (BC or AD)
w	One-digit (0-5) week of the month, where week 1 is the earliest set of four contiguous days ending on a Saturday
WW	Two-digit (01-53) ISO-8601 week of the year, where week 1 is the week containing January 4

Time symbols	Description
h	One- or two-digit (1-12) hour of the meridian (AM/PM)
hh	Zero-padded 2 digit (01-12) hour of the meridian (AM/PM)
k	One- or two-digit (0-11) hour of the meridian (AM/PM)
kk	Two-digit (00-11) hour of the meridian (AM/PM)
H	One- or two-digit (0-23) hour of the day
HH	Zero-padded, two-digit (00-23) hour of the day
K	One- or two-digit (1-24) hour of the day
KK	Zero-padded, two-digit (01-24) hour of the day
M	One- or two-digit (0-59) minute of the hour
MM	Zero-padded, two-digit (00-59) minute of the hour
S	One- or two-digit (0-59) second of the minute
SS	Zero-padded, two-digit (00-59) second of the minute
FFF	Three- digit (000-999) thousandth of the second
A	Meridian (AM or PM)
z	ISO-8601 time-zone format (for example, Z, +0500, -0030, -01, +0100)
zz	Alternative ISO-8601 time-zone format (for example, Z, +05:00, -00:30, -01, +01:00)
Z	Abbreviated time-zone name, for example, GMT, GMT+05:00, GMT-00:30, EST, PDT

Note: The h and H patterns reflect 12 and 24-hour clocks respectively.

Reserved symbols

The following symbols have special meanings and cannot be used as literal text.

?	Input parsing: The symbol matches any one character. Output formatting: The symbol becomes a space.
*	Input parsing: The symbol matches 0 or Unicode white space characters. Output formatting: The symbol becomes a space.
+	Input parsing: The symbol matches one or more Unicode white space characters. Output formatting: The symbol becomes a space.

Date

Returns the current system date as the number of days since the epoch.

Syntax

```
Date()
```

Parameters

None

Examples

The following expression is an example of using the Date function:

Expression	Returns
`Date()`	`37875` (the number of days from the epoch to September 12, 2003)

Date2Num

Returns the number of days since the epoch, given a date string.

Syntax

```
Date2Num(d [, f [, k ]])
```

Parameters

Parameter	Description
d	A date string in the format supplied by *f* that also conforms to the locale given by *k*.
f (*optional*)	A date format string. If *f* is omitted, the default date format `MMM D, YYYY` is used.
k (*optional*)	A locale identifier string that conforms to the locale naming standards. If *k* is omitted (or is invalid), the ambient locale is used.

The function returns a value of `0` if any of the following conditions are true:

- The format of the given date does not match the format specified in the function.
- Either the locale or date format supplied in the function is invalid.

Insufficient information is provided to determine a unique day since the epoch (that is, any information regarding the date is missing or incomplete).

Examples

The following expressions are examples of using the Date2Num function:

Expression	Returns
`Date2Num("Mar 15, 1996")`	`35138`
`Date2Num("1/1/1900", "D/M/YYYY")`	`1`
`Date2Num("03/15/96", "MM/DD/YY")`	`35138`
`Date2Num("Aug 1,1996", "MMM D, YYYY")`	`35277`
`Date2Num("96-08-20", "YY-MM-DD", "fr_FR")`	`35296`
`Date2Num("1/3/00", "D/M/YY") - Date2Num("1/2/00", "D/M/YY")`	`29`

DateFmt

Returns a date format string, given a date format style.

Syntax

```
DateFmt([n [, k ]])
```

Parameters

Parameter	Description
n (*optional*)	An integer identifying the locale-specific time format style as follows: • 0 (Default style) • 1 (Short style) • 2 (Medium style) • 3 (Long style) • 4 (Full style) If *n* is omitted (or is invalid), the default style value 0 is used.
k (*optional*)	A locale identifier string that conforms to the locale naming standards. If *k* is omitted (or is invalid), the ambient locale is used.

Examples

The following expressions are examples of using the DateFmt function:

Expression	Returns
`DateFmt()`	`MMM D, YYYY` This is the default date format.
`DateFmt(1)`	`M/D/YY`
`DateFmt(2, "fr_CA")`	`YY-MM-DD`
`DateFmt(3, "de_DE")`	`D. MMMM YYYY`
`DateFmt(4, "fr_FR")`	`EEEE D' MMMM YYYY`

IsoDate2Num

Returns the number of days since the epoch began, given a valid date string.

Syntax

```
IsoDate2Num(d)
```

Parameters

Parameter	Description
d	A valid date string.

Examples

The following expressions are examples of using the IsoDate2Num function:

Expression	Returns
`IsoDate2Num("1900")`	`1`
`IsoDate2Num("1900-01")`	`1`
`IsoDate2Num("1900-01-01")`	`1`
`IsoDate2Num("19960315T20:20:20")`	`35138`
`IsoDate2Num("2000-03-01") - IsoDate2Num("20000201")`	`29`

IsoTime2Num

Returns the number of milliseconds since the epoch, given a valid time string.

Syntax

```
IsoTime2Num(d)
```

Parameters

Parameter	Description
d	A valid time string.

Examples

The following expressions are examples of using the IsoTime2Num function:

Expression	Returns
`IsoTime2Num("00:00:00Z")`	`1`, for a user in the Eastern Time (ET) zone.
`IsoTime2Num("13")`	`64800001`, for a user located in Boston, U.S.
`IsoTime2Num("13:13:13")`	`76393001`, for a user located in California.
`IsoTime2Num("19111111T131313+01")`	`43993001`, for a user located in the Eastern Time (ET) zone.

LocalDateFmt

Returns a localized date format string, given a date format style.

Syntax

```
LocalDateFmt([n [, k ]])
```

Parameters

Parameter	Description
`n` (*optional*)	An integer identifying the locale-specific time format style as follows: • 0 (Default style) • 1 (Short style) • 2 (Medium style) • 3 (Long style) • 4 (Full style) If `n` is omitted (or is invalid), the default style value 0 is used.
`k` (*optional*)	A locale identifier string that conforms to the locale naming standards. If `k` is omitted (or is invalid), the ambient locale is used.

Examples

The following expressions are examples of the LocalDateFmt function:

Expression	Returns
`LocalDateFmt(1, "de_DE")`	`tt.MM.uu`
`LocalDateFmt(2, "fr_CA")`	`aa-nn-jj`
`LocalDateFmt(3, "de_CH")`	`t. MMMM uuuu`
`LocalDateFmt(4, "fr_FR")`	`EEEE j nnnn aaaa`

LocalTimeFmt

Returns a localized time format string, given a time format style.

Syntax

```
LocalTimeFmt([n [, k ]])
```

Parameters

Parameter	Description
`n` (*Optional*)	An integer identifying the locale-specific time format style as follows: • 0 (Default style) • 1 (Short style) • 2 (Medium style) • 3 (Long style) • 4 (Full style) If `n` is omitted (or is invalid), the default style value `0` is used.
`k` (*Optional*)	A locale identifier string that conforms to the locale naming standards. If `k` is omitted (or is invalid), the ambient locale is used.

Examples

The following expressions are examples of using the LocalTimeFmt function:

Expression	Returns
`LocalTimeFmt(1, "de_DE")`	`HH:mm`
`LocalTimeFmt(2, "fr_CA")`	`HH:mm:ss`
`LocalTimeFmt(3, "de_CH")`	`HH:mm:ss z`
`LocalTimeFmt(4, "fr_FR")`	`HH' h 'mm z`

Num2Date

Returns a date string, given a number of days since the epoch.

Syntax

```
Num2Date(n [,f [, k ]])
```

Parameters

Parameter	Description
n	An integer representing the number of days. If *n* is invalid, the function returns an error.
f (*Optional*)	A date format string. If you do not include a value for *f*, the function uses the default date format `MMM D, YYYY`.
k (*Optional*)	A locale identifier string that conforms to the locale naming standards. If you do not include a value for *k*, or if *k* is invalid, the function uses the ambient locale.

The function returns a value of `0` if any of the following conditions are true:

- The format of the given date does not match the format specified in the function.
- Either the locale or date format supplied in the function is invalid.

Insufficient information is provided to determine a unique day since the epoch (that is, any information regarding the date is missing or incomplete.

Examples

The following expressions are examples of using the Num2Date function:

Expression	Returns
`Num2Date(1, "DD/MM/YYYY")`	`01/01/1900`
`Num2Date(35139, "DD-MMM-YYYY", "de_DE")`	`16-Mrz-1996`
`Num2Date(Date2Num("Mar 15, 2000") - Date2Num("98-03-15", "YY-MM-DD", "fr_CA"))`	`Jan 1, 1902`

Num2GMTime

Returns a GMT time string, given a number of milliseconds from the epoch.

Syntax

```
Num2GMTime(n [,f [, k ]])
```

Parameters

Parameter	Description
n	An integer representing the number of milliseconds. If *n* is invalid, the function returns an error.
f (*Optional*)	A time format string. If you do not include a value for *f*, the function uses the default time format `H:MM:SS A`.
k (*Optional*)	A locale identifier string that conforms to the locale naming standards. If you do not include a value for *k*, or if *k* is invalid, the function uses the ambient locale.

The function returns a value of `0` if any of the following conditions are true:

- The format of the given time does not match the format specified in the function.
- Either the locale or time format supplied in the function is invalid.

Insufficient information is provided to determine a unique time since the epoch (that is, any information regarding the time is missing or incomplete.

Examples

The following expressions illustrate using the Num2GMTime function:

Expression	Returns
`Num2GMTime(1, "HH:MM:SS")`	`00:00:00`
`Num2GMTime(65593001, "HH:MM:SS Z")`	`18:13:13 GMT`
`Num2GMTime(43993001, TimeFmt(4, "de_DE"), "de_DE")`	`12.13 Uhr GMT`

Num2Time

Returns a time string, given a number of milliseconds from the epoch.

Syntax

```
Num2Time(n [,f [, k ]])
```

Parameters

Parameter	Description
n	An integer representing the number of milliseconds. If *n* is invalid, the function returns an error.
f (*Optional*)	A time format string. If you do not include a value for *f*, the function uses the default time format `H:MM:SS A`.
k (*Optional*)	A locale identifier string that conforms to the locale naming standards. If you do not include a value for *k*, or if *k* is invalid, the function uses the ambient locale.

The function returns a value of `0` if any of the following conditions are true:

- The format of the given time does not match the format specified in the function.
- Either the locale or time format supplied in the function is invalid.

Insufficient information is provided to determine a unique time since the epoch (that is, any information regarding the time is missing or incomplete.

Examples

The following expressions illustrate using the Num2Time function:

Expression	Returns
`Num2Time(1, "HH:MM:SS")`	`00:00:00` in Greenwich, England and `09:00:00` in Tokyo.
`Num2Time(65593001, "HH:MM:SS Z")`	`13:13:13` EST in Boston, U.S.
`Num2Time(65593001, "HH:MM:SS Z", "de_DE")`	`13:13:13 GMT-05:00` to a German-Swiss user in Boston, U.S.
`Num2Time(43993001, TimeFmt(4, "de_DE"), "de_DE")`	`13.13 Uhr GMT+01:00` to a user in Zurich, Austria.
`Num2Time(43993001, "HH:MM:SSzz")`	`13:13+01:00` to a user in Zurich, Austria.

Time

Returns the current system time as the number of milliseconds since the epoch.

Syntax

```
Time()
```

Parameters

None

Examples

The following expression is an example of using the Time function:

Expression	Returns
`Time()`	`71533235` at precisely 3:52:15 P.M. on September 15th, 2003 to a user in the Eastern Standard Time (EST) zone.

Time2Num

Returns the number of milliseconds since the epoch, given a time string.

Syntax

```
Time2Num(d [, f [, k ]])
```

Parameters

Parameter	Description
d	A time string in the format supplied by *f* that also conforms to the locale given by *k*.
f (*Optional*)	A time format string. If you do not include a value for *f*, the function uses the default time format `H:MM:SS A`.
k (*Optional*)	A locale identifier string that conforms to the locale naming standards. If you do not include a value for *k*, or if *k* is invalid, the function uses the ambient locale.

The function returns a value of `0` if any of the following conditions are true:

- The format of the given time does not match the format specified in the function.
- Either the locale or time format supplied in the function is invalid.

Insufficient information is provided to determine a unique time since the epoch (that is, any information regarding the time is missing or incomplete.

Examples

The following expressions illustrate using the Time2Num function:

Expression	Returns
`Time2Num("00:00:00 GMT", "HH:MM:SS Z")`	`1`
`Time2Num("1:13:13 PM")`	`76393001` to a user in California on Pacific Standard Time, and `76033001` when that same user is on Pacific Daylight Savings Time.
`Time2Num("13:13:13", "HH:MM:SS") - Time2Num("13:13:13 GMT", "HH:MM:SS Z")) / (60 * 60 * 1000)`	`8` to a user in Vancouver and `5` to a user in Ottawa when on Standard Time. On Daylight Savings Time, the returned values are `7` and `4`, respectively.
`Time2Num("13:13:13 GMT", "HH:MM:SS Z", "fr_FR")`	`47593001`

TimeFmt

Returns a time format, given a time format style.

Syntax

```
TimeFmt([n [, k ]])
```

Parameters

Parameter	Description
`n` (*Optional*)	An integer identifying the locale-specific time format style as follows: • 0 (Default style) • 1 (Short style) • 2 (Medium style) • 3 (Long style) • 4 (Full style) If you do not include a value for `n`, or if `n` is invalid, the function uses the default style value.
`k` (*Optional*)	A locale identifier string that conforms to the locale naming standards. If `k` is omitted (or is invalid), the ambient locale is used.

Examples

The following expressions are examples of using the TimeFmt function:

Expression	Returns
`TimeFmt()`	`h:MM:SS A`
`TimeFmt(1)`	`h:MM A`
`TimeFmt(2, "fr_CA")`	`HH:MM:SS`
`TimeFmt(3, "fr_FR")`	`HH:MM:SS Z`
`TimeFmt(4, "de_DE")`	`H.MM' Uhr 'Z`

6 Financial Functions

These functions perform a variety of interest, principal, and evaluation calculations related to the financial sector.

Functions

Apr

Returns the annual percentage rate for a loan.

Note: Interest rate calculation methods differ from country to country. This function calculates an interest rate based on U.S. interest rate standards.

Syntax

```
Apr(n1, n2, n3)
```

Parameters

Parameter	Description
n1	A numeric value or expression representing the principal amount of the loan.
n2	A numeric value or expression representing the payment amount on the loan.
n3	A numeric value or expression representing the number of periods in the loan's duration.

If any parameter is null, the function returns `null`. If any parameter is negative or 0, the function returns an error.

Note: FormCalc follows the IEEE-754 international standard when handling floating point numeric values. For more information, see "Number literals" on page 8.

Examples

The following expressions are examples of using the Apr function:

Expression	Returns
`Apr(35000, 269.50, 360)`	`0.08515404566` for a $35,000 loan repaid at $269.50 a month for 30 years.
`Apr(210000 * 0.75, 850 + 110, 25 * 26)`	`0.07161332404`
`Apr(-20000, 250, 120)`	Error
`Apr(P_Value, Payment, Time)`	This example uses variables in place of actual numeric values or expressions.

CTerm

Returns the number of periods needed for an investment earning a fixed, but compounded, interest rate to grow to a future value.

Note: Interest rate calculation methods differ from country to country. This function calculates an interest rate based on U.S. interest rate standards.

Syntax

```
CTerm(n1, n2, n3)
```

Parameters

Parameter	Description
n1	A numeric value or expression representing the interest rate per period.
n2	A numeric value or expression representing the future value of the investment.
n3	A numeric value or expression representing the amount of the initial investment.

If any parameter is null, the function returns null. If any parameter is negative or 0, the function returns an error.

Note: FormCalc follows the IEEE-754 international standard when handling floating point numeric values. For more information, see "Number literals" on page 8.

Examples

The following expressions are examples of using the CTerm function:

Expression	Returns
`CTerm(0.02, 1000, 100)`	`116.2767474515`
`CTerm(0.10, 500000, 12000)`	`39.13224648502`
`CTerm(0.0275 + 0.0025, 1000000, 55000 * 0.10)`	`176.02226044975`
`CTerm(Int_Rate, Target_Amount, P_Value)`	This example uses variables in place of actual numeric values or expressions.

FV

Returns the future value of consistent payment amounts made at regular intervals at a constant interest rate.

Note: Interest rate calculation methods differ from country to country. This function calculates an interest rate based on U.S. interest rate standards.

Syntax

```
FV(n1, n2, n3)
```

Parameters

Parameter	Description
n1	A numeric value or expression representing the payment amount.
n2	A numeric value or expression representing the interest per period of the investment.
n3	A numeric value or expression representing the total number of payment periods.

The function returns an error if either of the following conditions are true:

- Either of *n1* or *n3* are negative or 0.
- *n2* is negative.

If any of the parameters are null, the function returns null.

Note: FormCalc follows the IEEE-754 international standard when handling floating point numeric values. For more information, see "Number literals" on page 8.

Examples

The following expressions are examples of the FV function:

Expression	Returns
`FV(400, 0.10 / 12, 30 * 12)`	`904195.16991842445`. This is the value, after 30 years, of a $400 a month investment growing at 10% annually.
`FV(1000, 0.075 / 4, 10 * 4)`	`58791.96145535981`. This is the value, after 10 years, of a $1000 a month investment growing at 7.5% a quarter.
`FV(Payment[0], Int_Rate / 4, Time)`	This example uses variables in place of actual numeric values or expressions.

IPmt

Returns the amount of interest paid on a loan over a set period of time.

Note: Interest rate calculation methods differ from country to country. This function calculates an interest rate based on U.S. interest rate standards.

Syntax

```
IPmt(n1, n2, n3, n4, n5)
```

Parameters

Parameter	Description
n1	A numeric value or expression representing the principal amount of the loan.
n2	A numeric value or expression representing the annual interest rate of the investment.
n3	A numeric value or expression representing the monthly payment amount.
n4	A numeric value or expression representing the first month in which a payment will be made.
n5	A numeric value or expression representing the number of months for which to calculate.

The function returns an error if either of the following conditions are true:

- *n1*, *n2*, or *n3* are negative or 0.
- Either *n4* or *n5* are negative.

If any parameter is null, the function returns null. If the payment amount (*n3*) is less than the monthly interest load, the function returns `0`.

Note: FormCalc follows the IEEE-754 international standard when handling floating point numeric values. For more information, see "Number literals" on page 8.

Examples

The following expressions are examples of using the IPmt function:

Expression	Returns
`IPmt(30000, 0.085, 295.50, 7, 3)`	`624.8839283142`. The amount of interest repaid on a $30000 loan at 8.5% for the three months between the seventh month and the tenth month of the loan's term.
`IPmt(160000, 0.0475, 980, 24, 12)`	`7103.80833569485`. The amount of interest repaid during the third year of the loan.
`IPmt(15000, 0.065, 65.50, 15, 1)`	`0`, because the monthly payment is less than the interest the loan accrues during the month.

NPV

Returns the net present value of an investment based on a discount rate and a series of periodic future cash flows.

Note: Interest rate calculation methods differ from country to country. This function calculates an interest rate based on U.S. interest rate standards.

Syntax

```
NPV(n1, n2 [, ...])
```

Parameters

Parameter	Description
n1	A numeric value or expression representing the discount rate over a single period.
n2	A numeric value or expression representing a cash flow value, which must occur at the end of a period. It is important that the values specified in *n2* and beyond are in the correct sequence.

The function returns an error if *n1* is negative or 0. If any of the parameters are null, the function returns null.

Note: FormCalc follows the IEEE-754 international standard when handling floating point numeric values. For more information, see "Number literals" on page 8.

Examples

The following expressions are examples of using the NPV function:

Expression	Returns
`NPV(0.065, 5000)`	`4694.83568075117`, which is the net present value of an investment earning 6.5% per year that will generate $5000.
`NPV(0.10, 500, 1500, 4000, 10000)`	`11529.60863329007`, which is the net present value of an investment earning 10% a year that will generate $500, $1500, $4000, and $10,000 in each of the next four years.
`NPV(0.0275 / 12, 50, 60, 40, 100, 25)`	`273.14193838457`, which is the net present value of an investment earning 2.75% year that will generate $50, $60, $40, $100, and $25 in each of the next five months.

Pmt

Returns the payment for a loan based on constant payments and a constant interest rate.

Note: Interest rate calculation methods differ from country to country. This function calculates an interest rate based on U.S. interest rate standards.

Syntax

```
Pmt(n1, n2, n3)
```

Parameters

Parameter	Description
n1	A numeric value or expression representing the principal amount of the loan.
n2	A numeric value or expression representing the interest rate per period of the investment.
n3	A numeric value or expression representing the total number of payment periods.

The function returns an error if any parameter is negative or 0. If any parameter is null, the function returns null.

Note: FormCalc follows the IEEE-754 international standard when handling floating point numeric values. For more information, see "Number literals" on page 8.

Examples

The following expressions are examples of using the Pmt function:

Expression	Returns
`Pmt(150000, 0.0475 / 12, 25 * 12)`	`855.17604207164`, which is the monthly payment on a $150,000 loan at 4.75% annual interest, repayable over 25 years.
`Pmt(25000, 0.085, 12)`	`3403.82145169876`, which is the annual payment on a $25,000 loan at 8.5% annual interest, repayable over 12 years.

PPmt

Returns the amount of principal paid on a loan over a period of time.

Note: Interest rate calculation methods differ from country to country. This function calculates an interest rate based on US interest rate standards.

Syntax

```
PPmt(n1, n2, n3, n4, n5)
```

Parameters

Parameter	Description
n1	A numeric value or expression representing the principal amount of the loan.
n2	A numeric value or expression representing the annual interest rate.
n3	A numeric value or expression representing the amount of the monthly payment.
n4	A numeric value or expression representing the first month in which a payment will be made.
n5	A numeric value or expression representing the number of months for which to calculate.

The function returns an error if either of the following conditions are true:

- *n1*, *n2*, or *n3* are negative or 0.
- Either *n4* or *n5* is negative.

If any parameter is null, the function returns null. If the payment amount (*n3*) is less than the monthly interest load, the function returns `0`.

Note: FormCalc follows the IEEE-754 international standard when handling floating point numeric values. For more information, see "Number literals" on page 8.

Examples

The following expressions are examples of using the PPmt function:

Expression	Returns
`PPmt(30000, 0.085, 295.50, 7, 3)`	`261.6160716858`, which is the amount of principal repaid on a $30,000 loan at 8.5% for the three months between the seventh month and the tenth month of the loan's term.
`PPmt(160000, 0.0475, 980, 24, 12)`	`4656.19166430515`, which is the amount of principal repaid during the third year of the loan.
`PPmt(15000, 0.065, 65.50, 15, 1)`	`0`, because in this case the monthly payment is less than the interest the loan accrues during the month, therefore, no part of the principal is repaid.

PV

Returns the present value of an investment of periodic constant payments at a constant interest rate.

Note: Interest rate calculation methods differ from country to country. This function calculates an interest rate based on U.S. interest rate standards.

Syntax

```
PV(n1, n2, n3)
```

Parameters

Parameter	Description
n1	A numeric value or expression representing the payment amount.
n2	A numeric value or expression representing the interest per period of the investment.
n3	A numeric value or expression representing the total number of payment periods.

The function returns an error if either *n1* or *n3* is negative or 0. If any parameter is null, the function returns null.

Note: FormCalc follows the IEEE-754 international standard when handling floating point numeric values. For more information, see "Number literals" on page 8.

Examples

The following expressions are examples of using the PV function:

Expression	Returns
`PV(400, 0.10 / 12, 30 * 12)`	`45580.32799074439`. This is the value after 30 years, of a $400 a month investment growing at 10% annually.
`PV(1000, 0.075 / 4, 10 * 4)`	`58791.96145535981`. This is the value after ten years of a $1000 a month investment growing at 7.5% a quarter.
`PV(Payment[0], Int_Rate / 4, Time)`	This example uses variables in place of actual numeric values or expressions.

Rate

Returns the compound interest rate per period required for an investment to grow from present to future value in a given period.

Note: Interest rate calculation methods differ from country to country. This function calculates an interest rate based on U.S. interest rate standards.

Syntax

```
Rate(n1, n2, n3)
```

Parameters

Parameter	Description
n1	A numeric value or expression representing the future value of the investment.
n2	A numeric value or expression representing the present value of the investment.
n3	A numeric value or expression representing the total number of investment periods.

The function returns an error if any parameter is negative or 0. If any parameter is null, the function returns null.

Note: FormCalc follows the IEEE-754 international standard when handling floating point numeric values. For more information, see "Number literals" on page 8.

Examples

The following expressions are examples of using the Rate function:

Expression	Returns
`Rate(12000, 8000, 5)`	`0.0844717712` (or 8.45%), which is the interest rate per period needed for an $8000 present value to grow to $12,000 in five periods.
`Rate(10000, 0.25 * 5000, 4 * 12)`	`0.04427378243` (or 4.43%), which is the interest rate per month needed for the present value to grow to $10,000 in four years.
`Rate(Target_Value, Pres_Value[*], Term * 12)`	This example uses variables in place of actual numeric values or expressions.

Term

Returns the number of periods needed to reach a given future value from periodic constant payments into an interest bearing account.

Note: Interest rate calculation methods differ from country to country. This function calculates an interest rate based on U.S. interest rate standards.

Syntax

```
Term(n1, n2, n3)
```

Parameters

Parameter	Description
n1	A numeric value or expression representing the payment amount made at the end of each period.
n2	A numeric value or expression representing the interest rate per period of the investment.
n3	A numeric value or expression representing the future value of the investment.

The function returns an error if any parameter is negative or 0. If any parameter is null, the function returns null.

Note: FormCalc follows the IEEE-754 international standard when handling floating point numeric values. For more information, see "Number literals" on page 8.

Examples

The following expressions are examples of using the Term function:

Expression	Returns
`Term(475, .05, 1500)`	`3.00477517728` (or roughly 3), which is the number of periods needed to grow a payment of $475 into $1500, with an interest rate of 5% per period.

Expression	Returns
`Term(2500, 0.0275 + 0.0025, 5000)`	`1.97128786369`, which is the number of periods needed to grow payments of $2500 into $5000, with an interest rate of 3% per period.
`Rate(Inv_Value[0], Int_Rate + 0.0050, Target_Value)`	This example uses variables in place of actual numeric values or expressions. In this case, the first occurrence of the variable `Inv_Value` is used as the payment amount, half a percentage point is added to the variable `Int_Rate` to use as the interest rate, and the variable `Target_Value` is used as the future value of the investment.

7 Logical Functions

These functions are useful for testing and/or analyzing information to obtain a true or false result.

Functions

Choose

Selects a value from a given set of parameters.

Syntax

```
Choose(n, s1 [, s2 ...])
```

Parameters

Parameter	Description
n	The position of the value you want to select within the set. If this value is not a whole number, the function rounds *n* down to the nearest whole value. The function returns an empty string if either of the following conditions is true: • *n* is less than 1. • *n* is greater than the number of items in the set. If *n* is null, the function returns null.
s1	The first value in the set of values.
s2 (Optional)	Additional values in the set.

Examples

The following expressions are examples of using the Choose function:

Expression	Returns
`Choose(3, "Taxes", "Price", "Person", "Teller")`	`Person`
`Choose(2, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1)`	`9`
`Choose(Item_Num[0], Items[*])`	Returns the value within the set `Items` that corresponds to the position defined by the first occurrence of `Item_Num`.
`Choose(20/3, "A", "B", "C", "D", "E", "F", "G", "H")`	`F`

Exists

Determines whether the given parameter is an accessor to an existing object.

Syntax

```
Exists(v)
```

Parameters

Parameter	Description
v	A valid accessor value. If *v* is not an accessor, the function returns false (0).

Examples

The following expressions are examples of using the Exists function:

Expression	Returns
`Exists(Item)`	True (1) if the object `Item` exists and false (0) otherwise.
`Exists("hello world")`	False (0). The string is not an accessor.
`Exists(Invoice.Border.Edge[1].Color)`	True (1) if the object `Invoice` exists and has a `Border` property, which in turn has at least one `Edge` property, which in turn has a `Color` property. Otherwise, the function returns false (0).

HasValue

Determines whether the given parameter is an accessor with a non-null, non-empty, or non-blank value.

Syntax

```
HasValue(v)
```

Parameters

Parameter	Description
v	A valid accessor value. If *v* is not an accessor, the function returns false (0).

Examples

The following expressions are examples of using the HasValue function.

Expression	Returns
HasValue(2)	True (1)
HasValue(" ")	False (0)
HasValue(Amount[*])	Error
HasValue(Amount[0])	Evaluates the first occurrence of Amount and returns true (1) if it is a non-null, non-empty, or non-blank value.

Oneof

Determines whether the given value is within a set.

Syntax

```
Oneof(s1, s2 [, s3 ...])
```

Parameters

Parameter	Description
s1	The position of the value you want to select within the set. If this value is not a whole number, the function rounds *s1* down to the nearest whole value.
s2	The first value in the set of values.
s3 (*Optional*)	Additional values in the set.

Examples

The following expressions are examples of using the Oneof function:

Expression	Returns
`Oneof(3, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1)`	True (`1`)
`Oneof("John", "Bill", "Gary", "Joan", "John", "Lisa")`	True (`1`)
`Oneof(3, 1, 25)`	False"(`0`)
`Oneof("loan", Fields[*])`	Verifies whether any occurrence of `Fields` has a value of `loan`.

Within

Determines whether the given value is within a given range.

Syntax

```
Within(s1, s2, s3)
```

Parameters

Parameter	Description
s1	The value to test for. If *s1* is a number, the ordering comparison is numeric. If *s1* is not a number, the ordering comparison uses the collating sequence for the current locale. For more information, see "Locales" on page 42. If *s1* is null, the function returns null.
s2	The lower bound of the test range.
s3	The upper bound of the test range.

Examples

The following expressions are examples of using the Within function:

Expression	Returns
`Within("C", "A", "D")`	True (1)
`Within(1.5, 0, 2)`	True (1)
`Within(-1, 0, 2)`	False (0)
`Within($, 1, 10)`	True (1) if the current value is between 1 and 10.

8 Miscellaneous Functions

Functions in this section do not fit within any other particular function category and are useful in a variety of applications.

Functions

Eval

Returns the value of a given form calculation.

Syntax

```
Eval(s)
```

Parameters

Parameter	Description

s	A valid string representing an expression or list of expressions.
	Note: The Eval function cannot refer to user-defined variables and functions. For example:
	`var s = "var t = concat(s, ""hello"")"`
	`eval(s)`
	In this case, the Eval function does not recognize `s`, and so returns an error. Any subsequent functions that make reference to the variable `s` also fail.

Examples

The following expressions are examples of using the Eval function:

Expression	Returns
`eval("10*3+5*4")`	`50`
`eval("hello")`	`error`

Null

Returns the null value. The null value means no value.

Definition

```
Null()
```

Parameters

None

Examples

The following expressions are examples of using the Null function:

Expression	Returns
`Null()`	*null*
`Null() + 5`	5
`Quantity = Null()`	Assigns *null* to the object Quantity.
`Concat("ABC", Null(), "DEF")`	ABCDEF See also "Concat" on page 87.

Ref

Returns a reference to an existing object.

Definition

`Ref(v)`

Parameters

Parameters	Description
v	A valid string representing an accessor, reference, method, or function. **Note:** If the given parameter is null, the function returns the null reference. For all other given parameters, the function generates an error exception.

Examples

The following expressions are examples of using the Ref function:

Expressions	Returns
`Ref("10*3+5*4")`	`10*3+5*4`
`Ref("hello")`	`hello`

UnitType

Returns the units of a unitspan. A unitspan is a string consisting of a number followed by a unit name.

Syntax

`UnitType(s)`

Parameters

Parameter	Description
s	A valid string containing a numeric value and a valid unit of measurement (unitspan). Recognized units of measurement are: • in", inches • mm, millimeters • cm, centimeters • pt, picas, points • mp, millipoints If *s* is invalid, the function returns `in`.

Examples

The following expressions are examples of using the UnitType function:

Expression	Results
`UnitType("36 in")`	`in`
`UnitType("2.54centimeters")`	`cm`
`UnitType("picas")`	`pt`
`UnitType("2.cm")`	`cm`
`UnitType("2.zero cm")`	`in`
`UnitType("kilometers")`	`in`
`UnitType(Size[0])`	Returns the measurement value of the first occurrence of `Size`.

UnitValue

Returns the numerical value of a measurement with its associated unitspan, after an optional unit conversion. A unitspan is a string consisting of a number followed by a valid unit of measurement.

Syntax

```
UnitValue(s1 [, s2 ])
```

Parameters

Parameters	Description
s1	A valid string containing a numeric value and a valid unit of measurement (unitspan). Recognized units of measurement are: • in, inches • mm, millimeters • cm, centimeters • pt, picas, points • mp, millipoints
s2 (*optional*)	A string containing a valid unit of measurement. The function converts the unitspan specified in `s1` to this new unit of measurement. If you do not include a value for `s2`, the function uses the unit of measurement specified in `s1`. If `s2` is invalid, the function converts `s1` into inches.

Examples

The following expressions are examples of using the UnitValue function:

Expression	Returns
`UnitValue("2in")`	`2`
`UnitValue("2in", "cm")`	`5.08`
`UnitValue("6", "pt")`	`432`
`UnitValue("A", "cm")`	`0`
`UnitValue(Size[2], "mp")`	Returns the measurement value of the third occurrence of `Size` converted into millipoints.
`UnitValue("5.08cm", "kilograms")`	`2`

9 String Functions

Functions in this section deal with the manipulation, evaluation, and creation of string values.

Functions

At

Locates the starting character position of a string within another string.

Syntax

`At(s1, s2)`

Parameters

Parameter	Description
s1	The source string.
s2	The search string. If *s2* is not a part of *s1*, the function returns `0`. If *s2* is empty, the function returns `1`.

Examples

The following expressions are examples of using the At function:

Expression	Returns
`At("ABCDEFGH", "AB")`	`1`
`At("ABCDEFGH", "F")`	`6`
`At(23412931298471, 29)`	`5` The first occurrence of `29` within the source string.
`At(Ltrim(Cust_Info[0]), "555")`	The location of the string `555` within the first occurrence of `Cust_Info`. See also "Ltrim" on page 94.

Concat

Returns the concatenation of two or more strings.

Syntax

```
Concat(s1 [, s2 ...])
```

Parameters

Parameter	Description
s1	The first string in the set.
s2 (*Optional*)	Additional strings to append to the set.

Examples

The following expressions are examples of using the Concat function:

Expression	Returns
`Concat("ABC", "DEF")`	`ABCDEF`
`Concat("Tony", Space(1), "Blue")`	`Tony Blue` See also "Space" on page 99.
`Concat("You owe ", WordNum(1154.67, 2), ".")`	`You owe One Thousand One Hundred Fifty-four Dollars And Sixty-seven Cents.` See also "WordNum" on page 105.

Decode

Returns the decoded version of a given string.

Syntax

```
Decode(s1 [, s2 ])
```

Parameters

Parameter	Description
s1	The string to decode.
s2 (*Optional*)	A string identifying the type of decoding to perform. The following strings are valid decoding strings: • url (URL decoding) • html (HTML decoding) • xml (XML decoding) If you do not include a value for *s2*, the function uses URL decoding.

Examples

The following expressions are examples of using the Decode function:

Expression	Returns
`Decode("&AElig;&Aacute;&Acirc;&Aacute; &Acirc;", "html")`	ÆÁÂÁÂ
`Decode("~!@#$%^&*()_+|`{"}[] <>?,./;':", "xml")`	`~!@#$%^&*()_+|`{""}[]<>?,./;':`

Encode

Returns the encoded version of a given string.

Syntax

```
Encode(s1 [, s2 ])
```

Parameters

Parameter	Description
s1	The string to encode.
s2 (*Optional*)	A string identifying the type of encoding to perform. The following strings are valid encoding strings: • url (URL encoding) • html (HTML encoding) • xml (XML encoding) If you do not include a value for *s2*, the function uses URL encoding.

Examples

The following expressions are examples of using the Encode function:

Expression	Returns
`Encode("""hello, world!""", "url")`	`%22hello,%20world!%22`
`Encode("ÁÂÃÄÅÆ", "html")`	`&#xc1;&#Xc2;&#Xc3;&#xc4;&#xc5;&#xc6;`

Format

Formats the given data according to the specified picture format string.

Syntax

```
Format(s1, s2 [, s3 ...])
```

Parameters

Parameter	Description
s1	The picture format string, which may be a locale-sensitive picture clause. See "Locales" on page 42.
s2	The source data to format. For date picture formats, the source data must be either an ISO date-time string or an ISO date string in one of two formats: • YYYY[MM[DD]] • YYYY[-MM[-DD]] For time picture formats, the source data must be either an ISO date-time string or an ISO time string in one of the following formats: • HH[MM[SS[.FFF][z]]] • HH[MM[SS[.FFF][+HH[MM]]]] • HH[MM[SS[.FFF][-HH[MM]]]] • HH[:MM[:SS[.FFF][z] • HH[:MM[:SS[.FFF][-HH[:MM]]]] • HH[:MM[:SS[.FFF][+HH[:MM]]]] For date-time picture formats, the source data must be an ISO date-time string. For numeric picture formats, the source data must be numeric. For text picture formats, the source data must be textual. For compound picture formats, the number of source data arguments must match the number of subelements in the picture.
s3 (*Optional*)	Additional source data to format.

Examples

The following expressions are examples of using the Format function:

Expression	Returns
`Format("MMM D, YYYY", "20020901")`	`Sep 1, 2002`
`Format("$9,999,999.99", 1234567.89)`	`$1,234,567.89` in the U.S. and `1 234 567,89` Euros in France.

Left

Extracts a specified number of characters from a string, starting with the first character on the left.

Syntax

```
Left(s, n)
```

Parameters

Parameter	Description
s	The string to extract from.
n	The number of characters to extract. If the number of characters to extract is greater than the length of the string, the function returns the whole string. If the number of characters to extract is 0 or less, the function returns the empty string.

Examples

The following expressions are examples of using the Left function:

Expression	Returns
`Left("ABCDEFGH", 3)`	`ABC`
`Left("Tony Blue", 5)`	`"Tony "`
`Left(Telephone[0], 3)`	The first three characters of the first occurrence of `Telephone`.
`Left(Rtrim(Last_Name), 3)`	The first three characters of `Last_Name`. See also "Rtrim" on page 98.

Len

Returns the number of characters in a given string.

Syntax

`Len(s)`

Parameters

Parameter	Description
s	The string to examine.

Examples

The following expressions are examples of using the Len function:

Expression	Returns
`Len("ABDCEFGH")`	`8`
`Len(4)`	`1`
`Len(Str(4.532, 6, 4))`	`6` See also "Str" on page 100.
`Len(Amount[*])`	The number of characters in the first occurrence of `Amount`.

Lower

Converts all uppercase characters within a specified string to lowercase characters.

Syntax

```
Lower(s, [, k ])
```

Parameters

Parameter	Description
s	The string to convert.
k (*Optional*)	A string representing a valid locale. If you do not include a value for *k*, the function uses the ambient locale. See also "Locales" on page 42. **Note:** This function only converts the Unicode characters U+41 through U+5A (of the ASCII character set) as well as the characters U+FF21 through U+FF3A (of the fullwidth character set)

Examples

The following expressions are examples of using the Lower function:

Expression	Returns
`Lower("ABC")`	`abc`
`Lower("21 Main St.")`	`21 main st.`
`Lower(15)`	`15`
`Lower(Address[0])`	This example converts the first occurrence of `Address` to all lowercase letters.

Ltrim

Returns a string with all leading white space characters removed.

White space characters include the ASCII space, horizontal tab, line feed, vertical tab, form feed, carriage return, and the Unicode space characters (Unicode category Zs).

Syntax

```
Ltrim(s)
```

Parameters

Parameter	Description
s	The string to trim.

Examples

The following expressions are examples of using the Ltrim function:

Expression	Returns
`Ltrim("          ABCD")`	`"ABCD"`
`Ltrim(Rtrim("          Tony Blue          "))`	`"Tony Blue"` See also "Rtrim" on page 98.
`Ltrim(Address[0])`	Removes any leading white space from the first occurrence of `Address`.

Parse

Analyzes the given data according to the given picture format.

Parsing data successfully results in one of the following values:

- Date picture format: An ISO date string of the form YYYY-MM-DD.
- Time picture format: An ISO time string of the form HH:MM:SS.
- Date-time picture format: An ISO date-time string of the form YYYY-MM-DDTHH:MM:SS.
- Numeric picture format: A number.
- Text pictures: Text.

Syntax

```
Parse(s1, s2 )
```

Parameters

Parameter	Description
s1	A valid date or time picture format string. For more information on date and time formats, see "Structuring dates and times" on page 42.
s2	The string data to parse.

Examples

The following expressions are examples of using the Parse function:

Expression	Returns
`Parse("MMM D, YYYY", "Sep 1, 2002")`	`2002-09-01`
`Parse("$9,999,999.99", "$1,234,567.89")`	`1234567.89` in the U.S.

Replace

Replaces all occurrences of one string with another within a specified string.

Syntax

```
Replace(s1, s2 [, s3 ])
```

Parameters

Parameter	Description
s1	A source string.
s2	The string to replace.
s3 (*Optional*)	The replacement string. If you do not include a value for *s3*, or if *s3* is null, the function uses an empty string.

Examples

The following expressions are examples of using the Replace function:

Expression	Returns
`Replace("Tony Blue", "Tony", "Chris")`	`Chris Blue`
`Replace("ABCDEFGH", "D")`	`ABCEFGH`
`Replace("ABCDEFGH", "d")`	`ABCDEFGH`
`Replace(Comments[0], "recieve", "receive")`	Correctly updates the spelling of the word `receive` in the first occurrence of `Comments`.

Right

Extracts a number of characters from a given string, beginning with the last character on the right.

Syntax

```
Right(s, n )
```

Parameters

Parameter	Description
s	The string to extract.
n	The number of characters to extract. If *n* is greater than the length of the string, the function returns the whole string. If *n* is 0 or less, the function returns an empty string.

Examples

The following expressions are examples of using the Right function:

Expression	Returns
`Right("ABCDEFGH", 3)`	`FGH`
`Right("Tony Blue", 5)`	`" Blue"`
`Right(Telephone[0], 7)`	The last seven characters of the first occurrence of `Telephone`.
`Right(Rtrim(CreditCard_Num), 4)`	The last four characters of `CreditCard_Num`. See also "Rtrim" on page 98.

Rtrim

Returns a string with all trailing white space characters removed.

White space characters include the ASCII space, horizontal tab, line feed, vertical tab, form feed, carriage return, and the Unicode space characters (Unicode category Zs).

Syntax

```
Rtrim(s )
```

Parameters

Parameter	Description
s	The string to trim.

Examples

The following expressions are examples of using the Rtrim function:

Expression	Returns
`Rtrim("ABCD ")`	`"ABCD"`
`Rtrim("Tony Blue ")`	`"Tony Blue"`
`Rtrim(Address[0])`	Removes any trailing white space from the first occurrence of `Address`.

Space

Returns a string consisting of a given number of blank spaces.

Syntax

```
Space(n )
```

Parameters

Parameter	Description
n	The number of blank spaces.

Examples

The following expressions are examples of using the Space function:

Expression	Returns
`Space(5)`	`"     "`
`Space(Max(Amount[*]))`	A blank string with as many characters as the value of the largest occurrence of `Amount`. See also "Max" on page 36.
`Concat("Tony", Space(1), "Blue")`	`Tony Blue`

Str

Converts a number to a character string. FormCalc formats the result to the specified width and rounds to the specified number of decimal places.

Syntax

```
Str(n1 [, n2 [, n3 ]])
```

Parameters

Parameter	Description
n1	The number to convert.
n2 (*Optional*)	The maximum width of the string. If you do not include a value for *n2*, the function uses a value of `10` as the default width. If the resulting string is longer than *n2*, the function returns a string of * (asterisk) characters of the width specified by *n2*.
n3 (*Optional*)	The number of digits to appear after the decimal point. If you do not include a value for *n3*, the function uses 0 as the default precision.

Examples

The following expressions are examples of using the Str function:

Expression	Returns
`Str(2.456)`	`"          2"`
`Str(4.532, 6, 4)`	`4.5320`
`Str(234.458, 4)`	`" 234"`
`Str(31.2345, 4, 2)`	`****`
`Str(Max(Amount[*]), 6, 2)`	Converts the largest occurrence of `Amount` to a six-character string with two decimal places. See also "Max" on page 36.

Stuff

Inserts a string into another string.

Syntax

```
Stuff(s1, n1, n2 [, s2 ])
```

Parameters

Parameter	Description
s1	The source string.
n1	The position in *s1* to insert the new string *s2*. If *n1* is less than one, the function assumes the first character position. If *n1* is greater than length of *s1*, the function assumes the last character position.
n2	The number of characters to delete from string *s1*, starting at character position *n1*. If *n2* is less than or equal to 0, the function assumes 0 characters.
s2 (*Optional*)	The string to insert into *s1*. If you do not include a value for *s2*, the function uses the empty string.

Examples

The following expressions are examples of using the Stuff function:

Expression	Returns
`Stuff("TonyBlue", 5, 0, " ")`	`Tony Blue`
`Stuff("ABCDEFGH", 4, 2)`	`ABCFGH`
`Stuff(Address[0], Len(Address[0]), 0, "Street")`	This adds the word `Street` onto the end of the first occurrence of `Address`. See also "Len" on page 92.
`Stuff("members-list@myweb.com", 0, 0, "cc:"`	`cc:members-list@myweb.com`

Substr

Extracts a portion of a given string.

Syntax

```
Substr(s1, n1, n2 )
```

Parameters

Parameter	Description
s1	The source string.
n1	The position in string *s1* to start extracting. If *n1* is less than one, the function assumes the first character position. If *n1* is greater than length of *s1*, the function assumes the last character position.
n2	The number of characters to extract. If *n2* is less than or equal to 0, FormCalc returns an empty string. If *n1* + *n2* is greater than the length of *s1*, the function returns the substring starting at position *n1* to the end of *s1*.

Examples

The following expressions are examples of using the Substr function:

Expression	Returns
`Substr("ABCDEFG", 3, 4)`	`CDEF`
`Substr(3214, 2, 1)`	`2`
`Substr(Last_Name[0], 1, 3)`	Returns the first three characters from the first occurrence of `Last_Name`.
`Substr("ABCDEFG", 5, 0)`	`""`
`Substr("21 Waterloo St.", 4, 5)`	`Water`

Uuid

Returns a Universally Unique Identifier (UUID) string to use as an identification method.

Syntax

```
Uuid([n ])
```

Parameters

Parameter	Description
n	A number identifying the format of the UUID string. Valid numbers are: • 0 (default value): UUID string only contains hex octets. • 1: UUID string contains dash characters separating the sequences of hex octets at fixed positions. If you do not include a value for *n*, the function uses the default value.

Examples

The following expressions are examples of the Uuid function:

Expression	Returns
`Uuid()`	A value such as `3c3400001037be8996c400a0c9c86dd5`
`Uuid(0)`	A value such as `3c3400001037be8996c400a0c9c86dd5`
`Uuid(1)`	A value such as `1a3ac000-3dde-f352-96c4-00a0c9c86dd5`
`Uuid(7)`	A value such as `1a3ac000-3dde-f352-96c4-00a0c9c86dd5`

Upper

Converts all lowercase characters within a string to uppercase.

Syntax

```
Upper(s [, k ])
```

Parameters

Parameter	Description
s	The string to convert.
k (*Optional*)	A string representing a valid locale. If you do not include a value for *k*, the ambient locale is used. See also "Locales" on page 42. **Note:** This function only converts the Unicode characters U+61 through U+7A (of the ASCII character set) as well as the characters U+FF41 through U+FF5A (of the fullwidth character set).

Examples

The following expressions are examples of using the Upper function:

Expression	Returns
`Upper("abc")`	`ABC`
`Upper("21 Main St.")`	`21 MAIN ST.`
`Upper(15)`	`15`
`Upper(Address[0])`	This example converts the first occurrence of `Address` to all uppercase letters.

WordNum

Returns the English text equivalent of a given number.

Syntax

```
WordNum(n1 [, n2 [, k ]])
```

Parameters

Parameter	Description
n1	The number to convert. If any of the following statements is true, the function returns * (asterisk) characters to indicate an error: • *n1* is not a number. • The integral value of *n1* is negative. • The integral value of *n1* is greater than 922,337,203,685,477,550.
n2 (*Optional*)	A number identifying the formatting option. Valid numbers are: • 0 (default value): The number is converted into text representing the simple number. • 1: The number is converted into text representing the monetary value with no fractional digits. • 2: The number is converted into text representing the monetary value with fractional digits. If you do not include a value for *n2*, the function uses the default value (0).
k (*Optional*)	A string representing a valid locale. If you do not include a value for *k*, the function uses the ambient locale. See also "Locales" on page 42. **Note:** As of this release, it is not possible to specify a locale identifier other than English for this function.

Examples

The following expressions are examples of using the WordNum function.

Expression	Returns
`WordNum(123.45)`	`One Hundred and Twenty-three Dollars`
`WordNum(123.45, 1)`	`One Hundred and Twenty-three Dollars`
`WordNum(1154.67, 2)`	`One Thousand One Hundred Fifty-four Dollars And Sixty-seven Cents`
`WordNum(43, 2)`	`Forty-three Dollars And Zero Cents`
`WordNum(Amount[0], 2)`	This example uses the first occurrence of `Amount` as the conversion number.

10 URL Functions

These functions deal with the sending and receiving of information, including content types and encoding data, to any accessible URL locations.

Functions

Get

Downloads the contents of the given URL.

Syntax

`Get(s)`

Parameters

Parameter	Description
s	The URL to download. If the function is unable to download the URL, it returns an error.

Examples

The following expressions are examples of using the Get function:

Expression	Returns
`Get("http://www.myweb.com/data/mydata.xml")`	XML data taken from the specified file.
`Get("ftp://ftp.gnu.org/gnu/GPL")`	The contents of the GNU Public License.
`Get("http://intranet?sql=SELECT+*+FROM+projects+FOR+XML+AUTO,+ELEMENTS")`	The results of a SQL query to the specified website.

Post

Posts the given data to the specified URL.

Syntax

```
Post(s1, s2 [, s3 [, s4 [, s5 ]]])
```

Parameters

Parameter	Description
s1	The URL to post to.
s2	The data to post. If the function is unable to post the data, it returns an error.
s3 (*Optional*)	A string containing the content type of the data to post. Valid content types include: • application/octet-stream (default value) • text/html • text/xml • text/plain • multipart/form-data • application/x-www-form-urlencoded • Any other valid MIME type If you do not include a value for *s3*, the function sets the content type to the default value. The application is responsible for ensuring that the data to post uses the correct format according to the specified content type.
s4 (*Optional*)	A string containing the name of the code page used to encode the data. Valid code page names include: • UTF-8 (default value) • UTF-16 • ISO-8859-1 • Any character encoding listed by the Internet Assigned Numbers Authority (IANA) If you do not include a value for *s4*, the function sets the code page to the default value. The application is responsible for ensuring that encoding of the data to post matches the specified code page.
s5 (*Optional*)	A string containing any additional HTTP headers to be included with the posting of the data. If you do not include a value for *s5*, the function does not include an additional HTTP header in the post. SOAP servers usually require a SOAPAction header when posting to them.

Examples

The following expressions are examples of using the Post function:

Expression	Returns
`Post("http://tools_build/scripts/jfecho.cgi", "user=joe&passwd=xxxxx&date=27/08/2002", "application/x-www-form-urlencoded")`	Posts some URL encoded login data to a server and returns that server's acknowledgement page.
`Post("http://www.nanonull.com/TimeService/TimeService.asmx/getLocalTime", "<?xml version='1.0' encoding='UTF-8'?><soap:Envelope><soap:Body><getLocalTime/></soap:Body></soap:Envelope>", "text/xml", "utf-8", "http://www.Nanonull.com/TimeService/getLocalTime")`	Posts a SOAP request for the local time to some server, expecting an XML response back.

Put

Uploads the given data to the specified URL.

Syntax

```
Put(s1, s2 [, s3 ])
```

Parameters

Parameter	Description
s1	The URL to upload.
s2	The data to upload. If the function is unable to upload the data, it returns an error.
s3 (*Optional*)	A string containing the name of the code page used to encode the data. Valid code page names include: • UTF-8 (default value) • UTF-16 • ISO8859-1 • Any character encoding listed by the Internet Assigned Numbers Authority (IANA) If you do not include a value for *s3*, the function sets the code page to the default value. The application is responsible for ensuring that encoding of the data to upload matches the specified code page.

Examples

The following expressions is an example of using the Put function:

Expression	Returns
`Put("ftp://www.example.com/pub/fubu.xml", "<?xml version='1.0' encoding='UTF-8'?><msg>hello world!</msg>")`	Nothing if the FTP server has permitted the user to upload some XML data to the file `pub/fubu.xml`. Otherwise, this function returns an error.

Index

L

M

N

O

P

R

S

T

U

V

W