FormCalc User Reference

Adobe® LiveCycle® Designer ES

July 2008

Version 8.2

Adobe® LiveCycle® Designer ES 8.2 FormCalc User Reference for Microsoft® Windows®
Edition 3.0, July 2008

Contents

Preface

Adobe® LiveCycle® Designer ES 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, LiveCycle Designer ES 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 LiveCycle Designer ES who want 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 LiveCycle Designer ES.

Related documentation

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

If you require more technical information about FormCalc, refer to the *Adobe XML Forms Architecture (XFA) Specification, version 2.4*, available from http://partners.adobe.com/public/developer/xml/index_arch.html.

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 LiveCycle Designer ES

Within LiveCycle Designer ES, 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 forms. However, if you are using a server-based process, such as LiveCycle Forms ES, 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 insert two 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"
```

Commenting all FormCalc calculations on an event

Commenting all of the FormCalc calculations for a particular event generates an error when you preview your form in the Preview PDF tab or when you view the final PDF. Each FormCalc calculation is required to return a value, and FormCalc does not consider comments to be values.

To prevent the commented FormCalc code from returning an error, you must do one of the following actions:

- Remove the commented code from the event
- Add an expression that returns a value to the FormCalc code on the event

To prevent the value of the expression from producing unwanted results on your form, use one of the following types of expressions:

- A simple expression consisting of a single character, as shown in the following example:

```
// First_Name="Tony"
// Initial="C"
// Last_Name="Blue"
//
// The simple expression below sets the value of the event to zero.
0
```

- An assignment expression that retains the value of the object. Use this type of expression if your commented FormCalc code is located on the calculate event to prevent the actual value of the object from being altered, as shown in the following example:

```
// First_Name="Tony"
// Initial="C"
// Last_Name="Blue"
//
// The assignment expression below sets the value of the current
// field equal to itself.
$.rawValue = $.rawValue
```

Keywords

Keywords in FormCalc are reserved words and are case-insensitive. Keywords are used as 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 reference syntax 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 reference syntax, see “Reference Syntax” on page 24.

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.

While expressions

A while expression is an iterative statement or loop that evaluates a given simple expression. If the result of the evaluation is true (1), FormCalc repeatedly examines the do condition and returns the results of the expression lists. If the result is false (0), then control passes to the next statement.

A while expression is particularly well-suited to situations in which conditional repetition is needed. Conversely, situations in which unconditional repetition is needed are often best dealt with using a for expression.

Expression	Syntax	Returns

While	`while ( simple expression ) do`
	`  expression list`
	`endwhile`
		The result of the list of expressions associated with the do condition.

In the following example, the values of the elements are added to a drop-down list from an XML file using the addItem method for all of the XML elements listed under list1 that are not equal to 3:

```
var List = ref(xfa.record.lists.list1)
```

```
var i = 0
while ( List.nodes.item(i+1).value ne "3")do
$.addItem (List.nodes.item(i).value,List.nodes.item(i+1).value)
i = i + 2
endwhile
```

For expressions

A for expression is a conditionally iterative statement or loop.

A for expression is particularly well-suited to looping situations in which unconditional repetition is needed. Conversely, situations in which conditional repetition is needed are often best dealt with using a while expression.

The value of the for expression is the value of the last evaluation list that was evaluated, or false (0).

The for condition initializes a FormCalc variable, which controls the looping action.

In the upto variant, the value of the loop variable will iterate from the start expression to the end expression in step expression increments. If you omit the step expression, the step increment defaults to 1.

In the downto variant, the value of the loop variable iterates from the start expression to the end expression in step expression decrements. If the step expression is omitted, the step decrements defaults to -1.

Iterations of the loop are controlled by the end expression value. Before each iteration, the end expression is evaluated and compared to the loop variable. If the result is true (1), the expression list is evaluated. After each evaluation, the step expression is evaluated and added to the loop variable.

Before each iteration, the end expression is evaluated and compared to the loop variable. In addition, after each evaluation of the do condition, the step expression is evaluated and added to the loop variable.

A for loop terminates when the start expression has surpassed the end expression. The start expression can surpass the end expression in either an upwards direction, if you use upto, or in a downward direction, if you use downto.

Expression	Syntax	Returns

For	`for variable = start expression`
	`(upto | downto ) end expression`
	`(step step expression ) do`
	`expression list`
	`endfor`
	
	Note: The start, end, and step expressions must all be simple expressions.
		The result of the list of expressions associated with the do condition.

In the following example, the values of the elements are added to a drop-down list from an XML file using the addItem method for all of the XML elements listed under list1:

```
var List = ref(xfa.record.lists.list1)
for i=0 upto List.nodes.length - 1 step 2 do
$.addItem (List.nodes.item(i).value,"")
endfor
```

Foreach expressions

A foreach expression iterates over the expression list for each value in its argument list.

The value of the foreach expression is the value of the last expression list that was evaluated, or zero (0), if the loop was never entered.

The in condition, which is executed only once (after the loop variable has been declared) controls the iteration of the loop. Before each iteration, the loop variable is assigned successive values from the argument list. The argument list cannot be empty.

Expression	Syntax	Returns

Foreach	`foreach variable in( argument list )do`
	`  expression list`
	`endfor`
	
	Note: Use a comma (,) to separate more than one simple expression in the argument list.
		The value of the last expression list that was evaluated, or zero(0), if the loop was never entered.

In the following example, only the values of the "display" XML elements are added to the foreach drop-down list.

```
foreach Item in (xfa.record.lists.list1.display[*]) do
$.addItem(Item,"")
endfor
```

Break expressions

A break expression causes an immediate exit from the innermost enclosing while, for, or foreach expression loop. Control passes to the expression following the terminated loop.

The value of the break expression is always the value zero (0).

Expression	Syntax	Returns
Break	`break`	Passes control to the expression following the terminated loop.

In the following example, an if condition is placed in the while loop to check whether the current value is equal to "Display data for 2". If true, the break executes and stops the loop from continuing.

```
var List = ref(xfa.record.lists.list1)
var i=0
while (List.nodes.item(i+1).value ne "3") do
$.addItem(List.nodes.item(i).value,List.nodes.item(i+1).value)
i = i + 2
if (List.nodes.item(i) eq "Display data for 2" then
break
endif
endwhile
```

Continue expressions

A continue expression causes the next iteration of the innermost enclosing while, for, or foreach loop.

The value of the continue expression is always the value zero (0).

Expression	Syntax	Returns
Continue	`continue`	When used in a while expression, control is passed to the while condition. When used in a for expression, control is passed to the step expression.

The object of the following example is to populate the drop-down list with values from the XML file. If the value of the current XML element is "Display data for 3," then the while loop exits via the break expression. If the value of the current XML element is "Display data for 2", then the script adds 2 to the variable `i` (which is the counter) and immediately the loop moves on to its next cycle. The last two lines are ignored when the value of the current XML element is "Display data for 2".

```
var List = ref(xfa.record.lists.list1)
var i = 0
while (List.nodes.item(i+1).value ne "5") do
if (List.nodes.item(i) eq "Display data for 3") then
break
endif
if (List.nodes.item(i) eq "Display data for 2" then
i=i+2
continue
endif
$.addItem(List.nodes.item(i).value,List.nodes.item(i+1).value)
i=i+2
endwhile
```

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.

Reference Syntax

FormCalc provides access to form design object properties and values using a reference syntax. The following example demonstrates both assigning and retrieving object values:

```
Invoice_Total.rawValue = Invoice_SubTotal.rawValue * (8 / 100)
```

In this case the reference syntax `Invoice_Total` assigns the value of `Invoice_SubTotal * (8 / 100)` to the field `Invoice_Total`.

In the context of form design, a fully qualified reference syntax enables access to all the objects on a form design.

To make accessing object properties and values easier, FormCalc includes shortcuts to reduce the effort required to create references. The following table outlines the reference syntax shortcuts for FormCalc.

Notation	Description
`$`	Refers to the current field or object, as shown in this example: `$ = "Tony Blue"` The above example sets the value of the current field or object to `Tony Blue`.

`$data`	Represents the root of the data model `xfa.datasets.data`. For example,
	`$data.purchaseOrder.total`
	is equivalent to
	`xfa.datasets.data.purchaseOrder.total`

`$event`	Represents the current form object event. For example,
	`$event.name`
	is equivalent to
	`xfa.event.name`

`$form`	Represents the root of the form model `xfa.form`. For example,
	`$form.purchaseOrder.tax`
	is equivalent to stating
	`xfa.form.purchaseOrder.tax`

`$host`	Represents the host object. For example,
	`$host.messageBox("Hello world")`
	is equivalent to
	`xfa.host.messageBox("Hello world")`

`$layout`	Represents the root of the layout model `xfa.layout`. For example,
	`$layout.ready`
	is equivalent to stating
	`xfa.layout.ready`

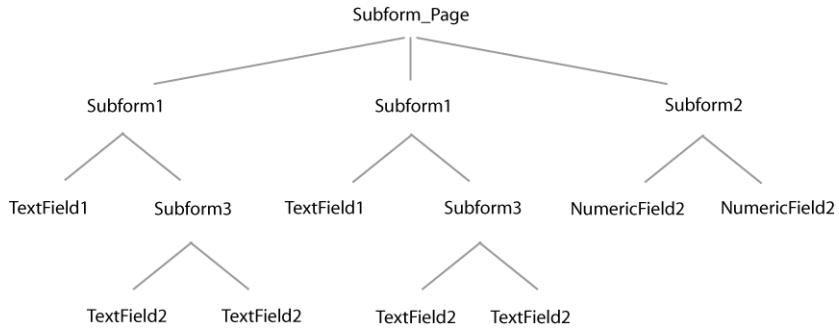
Notation	Description
`$record`	Represents the current record of a collection of data, such as from an XML file. For example, `$record.header.txtOrderedByCity` references the `txtOrderedByCity` node within the `header` node of the current XML data.

`$template`	Represents the root of the template model `xfa.template`. For example,
	`$template.purchaseOrder.item`
	is equivalent to
	`xfa.template.purchaseOrder.item`

`!`	Represents the root of the data model `xfa.datasets`. For example,
	`!data`
	is equivalent to
	`xfa.datasets.data`

*	Selects all form objects within a given container, such as a subform, regardless of name, or selects all objects that have a similar name. For example, the following expression selects all objects named `item` on a form: `xfa.form.form1.item[*]` **Note:** You can use the '*' (asterisk) syntax with JavaScript if it used with the `resolveNode` method. For more information about the `resolveNode` method, see *LiveCycle Designer ES Help*, or see the LiveCycle Designer ES Scripting Reference.

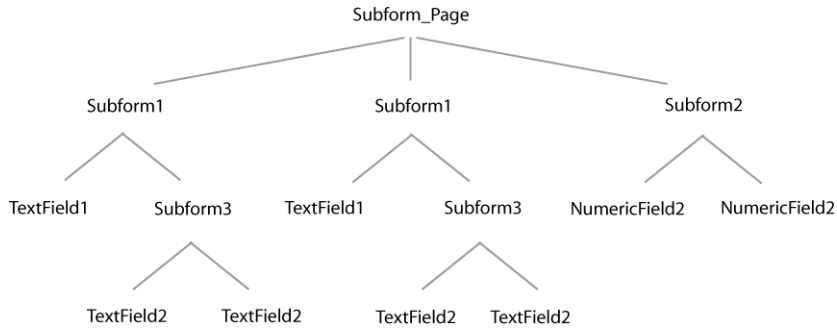
Notation	Description

`..`	You can use two dots at any point in your reference syntax to search for objects that are a part of any subcontainer of the current container object, such as a subform. For example, the expression `Subform_Page..Subform2` means locate the node `Subform_Page` (as usual) and find a descendant of `Subform_Page` called `Subform2`.
	 
	
	Using the example tree above,
	`Subform_Page..TextField2`
	is equivalent to
	`Subform_Page.Subform1[0].Subform3.TextField2[0]`
	because `TextField2[0]` is in the first `Subform1` node that FormCalc encounters on its search. As a second example,
	`Subform_Page..Subform3[*]`
	returns all four `TextField2` objects.
	Note: You can use the '..' (double period) syntax with JavaScript if it used with the `resolveNode` method. For more information about the `resolveNode` method, see *LiveCycle Designer ES Help*, or see the LiveCycle Designer ES Scripting Reference.

#	The number sign (#) notation is used to denote one of the following items in a reference syntax:
	• An unnamed object. For example, the following reference syntax accesses an unnamed subform:
	`xfa.form.form1.#subform`
	• Specify a property in a reference syntax if a property and an object have the same name. For example, the following reference syntax accesses the `name` property of a subform if the subform also contains a field named `name`:
	`xfa.form.form1.#subform.#name`
	Note: You can use the '#' (number sign) syntax with JavaScript if it used with the `resolveNode` method. For more information about the `resolveNode` method, see *LiveCycle Designer ES Help*, or see the LiveCycle Designer ES Scripting Reference.

Notation	Description

`[]`	The square bracket (`[ ]`) notation denotes the occurrence value of an object. To construct an occurrence value reference, place square brackets (`[ ]`) after an object name, and enclose within the brackets one of the following values:
	• `[ n ]`, where `n` is an absolute occurrence index number beginning at 0. An occurrence number that is out of range does not return a value. For example,
	`xfa.form.form1.#subform.Quantity[3]`
	`refers to the fourth occurrence of the Quantity object.`
	• `[ +/- n ]`, where `n` indicates an occurrence relative to the occurrence of the object making the reference. Positive values yield higher occurrence numbers, and negative values yield lower occurrence numbers. For example,
	`xfa.form.form1.#subform.Quantity[+2]`
	This reference 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 the `Quantity[2]` object , the reference would be the same as
	`xfa.template.Quantity[4]`
	If the computed index number is out of range, the reference returns an error.
	The most common use of this syntax is for locating the previous or next occurrence of a particular object. For example, every occurrence of the `Quantity` object (except the first) might use `Quantity[-1]` to get the value of the previous `Quantity` object.
	• `[*]` indicates multiple occurrences of an 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 a collection of objects. For example,
	`xfa.form.form1.#subform.Quantity[*]`
	This expression refers to all objects with a name of `Quantity` that are siblings to the first occurrence of `Quantity` found by the reference.
	Note: In language-specific forms for Arabic, Hebrew, Thai, and Vietnamese, the reference syntax is always on the right (even for right-to-left languages).

Notation	Description
[ ] (Continued)	 Using the tree for reference, these expressions return the following objects: • `Subform_Page.Subform1[*]` returns both `Subform1` objects. • `Subform_Page.Subform1.Subform3.TextField2[*]` returns two `TextField2` objects. `Subform_Page.Subform1` resolves to the first `Subform1` object on the left, and `TextField2[*]` evaluates relative to the `Subform3` object. • `Subform_Page.Subform1[*].TextField1` returns both of the `TextField1` instances. `Subform_Page.Subform1[*]` resolves to both `Subform1` objects, and `TextField1` evaluates relative to the `Subform1` objects. • `Subform_Page.Subform1[*].Subform3.TextField2[1]` returns the second and fourth `TextField2` objects from the left. `Subform_Page.Subform1[*]` resolves to both `Subform1` objects, and `TextField2[1]` evaluates relative to the `Subform3` objects. • `Subform_Page.Subform1[*].Subform3[*]` returns both instances of the `Subform3` object. • `Subform_Page.*` returns both `Subform1` objects and the `Subform2` object. • `Subform_Page.Subform2.*` returns the two instances of the `NumericField2` object. **Note:** You can use the '[]' (square bracket) syntax with JavaScript if it used with the `resolveNode` method. For more information about the `resolveNode` method, see *LiveCycle Designer ES Help*, or see the LiveCycle Designer ES Scripting Reference.

Property and method calls

LiveCycle Designer ES defines a variety of properties and methods for all objects on a form design. FormCalc provides access to these properties and methods and allows you to use them to modify the appearance and behavior of objects on your form. Similar to a function call, you invoke properties and methods by passing arguments to them in a specific order. The number and type of arguments in each property and method are specific to each object type.

Note: Different form design objects support different properties and methods. For a complete list of the properties and methods objects support, see LiveCycle Designer ES Scripting Reference.

Built-in 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 30.

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 34	Returns the absolute value of a numeric value or expression.	Arithmetic
"Apr" on page 60	Returns the annual percentage rate for a loan.	Financial
"At" on page 78	Locates the starting character position of a string within another string.	String
"Avg" on page 35	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 35	Returns the whole number greater than or equal to a given number.	Arithmetic
"Choose" on page 70	Selects a value from a given set of parameters.	Logical
"Concat" on page 79	Returns the concatenation of two or more strings.	String
"Count" on page 36	Evaluates a set of values and/or expressions and returns the number of non-null elements contained within the set.	Arithmetic
"CTerm" on page 61	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 51	Returns the current system date as the number of days since the epoch.	Date and Time
"Date2Num" on page 51	Returns the number of days since the epoch, given a date string.	Date and Time
"DateFmt" on page 52	Returns a date format string, given a date format style.	Date and Time
"Decode" on page 80	Returns the decoded version of a given string.	String
"Encode" on page 80	Returns the encoded version of a given string.	String
"Eval" on page 74	Returns the value of a given form calculation.	Miscellaneous
"Exists" on page 71	Determines whether the given parameter is a reference syntax to an existing object.	Logical
"Floor" on page 36	Returns the largest whole number that is less than or equal to the given value.	Arithmetic

Function	Description	Type
"Format" on page 81	Formats the given data according to the specified picture format string.	String
"FV" on page 62	Returns the future value of consistent payment amounts made at regular intervals at a constant interest rate.	Financial
"Get" on page 93	Downloads the contents of the given URL.	URL
"HasValue" on page 71	Determines whether the given parameter is an accessor with a non-null, non-empty, or non-blank value.	Logical
"IPmt" on page 63	Returns the amount of interest paid on a loan over a set period of time.	Financial
"IsoDate2Num" on page 53	Returns the number of days since the epoch, given an valid date string.	Date and Time
"IsoTime2Num" on page 53	Returns the number of milliseconds since the epoch, given a valid time string.	Date and Time
"Left" on page 82	Extracts a specified number of characters from a string, starting with the first character on the left.	String
"Len" on page 83	Returns the number of characters in a given string.	String
"LocalDateFmt" on page 54	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 84	Converts all uppercase characters within a specified string to lowercase characters.	String
"Ltrim" on page 84	Returns a string with all leading white space characters removed.	String
"Max" on page 37	Returns the maximum value of the non-null elements in the given set of numbers.	Arithmetic
"Min" on page 38	Returns the minimum value of the non-null elements of the given set of numbers.	Arithmetic
"Mod" on page 39	Returns the modulus of one number divided by another.	Arithmetic
"NPV" on page 64	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 75	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 72	Returns true (1) if a value is in a given set, and false (0) if it is not.	Logical
"Parse" on page 85	Analyzes the given data according to the given picture format.	String
"Pmt" on page 64	Returns the payment for a loan based on constant payments and a constant interest rate.	Financial
"Post" on page 93	Posts the given data to the specified URL.	URL
"PPmt" on page 65	Returns the amount of principal paid on a loan over a period of time.	Financial
"Put" on page 95	Uploads the given data to the specified URL.	URL
"PV" on page 66	Returns the present value of an investment of periodic constant payments at a constant interest rate.	Financial
"Rate" on page 67	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 75	Returns a reference to an existing object.	Miscellaneous
"Replace" on page 85	Replaces all occurrences of one string with another within a specified string.	String
"Right" on page 86	Extracts a number of characters from a given string, beginning with the last character on the right.	String
"Round" on page 40	Evaluates a given numeric value or expression and returns a number rounded to the given number of decimal places.	Arithmetic
"Rtrim" on page 87	Returns a string with all trailing white space characters removed.	String
"Space" on page 87	Returns a string consisting of a given number of blank spaces.	String
"Str" on page 88	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 89	Inserts a string into another string.	String
"Substr" on page 89	Extracts a portion of a given string.	String
"Sum" on page 41	Returns the sum of the non-null elements of a given set of numbers.	Arithmetic

Function	Description	Type
"Term" on page 68	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 57	Returns the current system time as the number of milliseconds since the epoch.	Date and Time
"Time2Num" on page 58	Returns the number of milliseconds since the epoch, given a time string.	Date and Time
"TimeFmt" on page 59	Returns a time format, given a time format style.	Date and Time
"UnitType" on page 76	Returns the units of a unitspan. A unitspan is a string consisting of a number followed by a unit name.	Miscellaneous
"UnitValue" on page 77	Returns the numeric value of a measurement with its associated unitspan, after an optional unit conversion.	Miscellaneous
"Upper" on page 91	Converts all lowercase characters within a string to uppercase.	String
"Uuid" on page 90	Returns a Universally Unique Identifier (UUID) string to use as an identification method.	String
"Within" on page 73	Returns true (1) if the test value is within a given range, and false (0) if it is not.	Logical
"WordNum" on page 92	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 9.

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 9.

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 9.

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 9.

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 9.

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 9.

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`

Expression	Returns
`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 38.

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 9.

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`

Expression	Returns
`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 37.

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 9.

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`

Expression	Returns
`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 37 and "Round" on page 40.

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 9.

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 62.
`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 9.

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 37, "Min" on page 38, and "Round" on page 40.

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 region). For the purposes of FormCalc, a *locale* defines the format of dates, times, numeric, and currency values relevant to a specific nation or region so that users can use the 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 consist of a language part, a country or region part, or both. The following table lists valid locales for this release of LiveCycle Designer ES.

Language	Country or Region	ISO Code
Arabic	United Arabian Emirates	ar_AE
Arabic	Bahrain	ar_BH
Arabic	Algeria	ar_DZ
Arabic	Egypt	ar_EG

Language	Country or Region	ISO Code
Arabic	Iraq	ar_IQ
Arabic	Jordan	ar_JO
Arabic	Kuwait	ar_KW
Arabic	Lebanon	ar_LB
Arabic	Libya	ar_LY
Arabic	Morocco	ar_MA
Arabic	Oman	ar_OM
Arabic	Qatar	ar_QA
Arabic	Saudi Arabia	ar_SA
Arabic	Sudan	ar_SD
Arabic	Syria	ar_SY
Arabic	Tunisia	ar_TN
Arabic	Yemen	ar_YE
Bulgarian	Bulgaria	bg_BG
Chinese	Hong Kong	zh_HK
Chinese	People's Republic of China (Simplified)	zh_CN
Chinese	Taiwan (Traditional)	zh_TW
Croatian	Croatia	hr_HR
Czech	Czech Republic	cs_CAZ
Danish	Denmark	da_DK
Dutch	Belgium	nl_BE
Dutch	Netherlands	nl_NL
English	Australia	en_AU
English	Canada	en_CA
English	India	en_IN
English	Ireland	en_IE
English	New Zealand	en_NZ
English	South Africa	en_ZA
English	United Kingdom	en_GB
English	United Kingdom Euro	en_GB_EURO

Language	Country or Region	ISO Code
English	United States of America	en_US
Estonian	Estonia	et_EE
Finnish	Finland	fi_FI
French	Belgium	fr_BE
French	Canada	fr_CA
French	France	fr_FR
French	Luxembourg	fr_LU
French	Switzerland	fr_CH
German	Austria	de_AT
German	Germany	de_DE
German	Luxembourg	de_LU
German	Switzerland	de_CH
Greek	Greece	el_GR
Hebrew	Israel	he_IL
Hungarian	Hungary	hu_HU
Indonesian	Indonesia	id_ID
Italian	Italy	it_IT
Italian	Switzerland	it_CH
Japanese	Japan	ja_JP
Korean	Republic of Korea	ko_KR
Korean	Korea Hanja	ko_KR_HANI
Latvian	Latvia	lv_LV
Lithuanian	Lithuania	lt_LT
Malay	Malaysia	ms_MY
Norwegian - Bokmal	Norway	nb_NO
Norwegian - Nynorsk	Norway	nn_NO
Polish	Poland	pl_PL
Portuguese	Brazil	pt_BR
Portuguese	Portugal	pt_PT
Romanian	Romania	ro_RO
Russian	Russia	ru_RU

Language	Country or Region	ISO Code
Serbo-Croatian	Bosnia and Herzegovina	sh_BA
Serbo-Croatian	Croatia	sh_HR
Serbo-Croatian	Serbia and Montenegro	sh_CS
Slovak	Slovakia	sk_SK
Slovenian	Slovenia	sl_SI
Spanish	Ecuador	es_EC
Spanish	El Salvador	es_SV
Spanish	Guatemala	es_GT
Spanish	Honduras	es_HN
Spanish	Nicaragua	es_NI
Spanish	Panama	es_PA
Spanish	Paraguay	es_PY
Spanish	Puerto Rico	es_PR
Spanish	Uruguay	es_UY
Spanish	Argentina	es_AR
Spanish	Bolivia	es_BO
Spanish	Chile	es_CL
Spanish	Columbia	es_CO
Spanish	Costa Rica	es_CR
Spanish	Dominican Republic	es_DO
Spanish	Mexico	es_MX
Spanish	Peru	es_PE
Spanish	Spain	es_ES
Spanish	Venezuela	es_VE
Swedish	Sweden	sv_SE
Thai	Thailand	th_TH
Thai	Thailand Traditional	th_TH_TH
Turkish	Turkey	tr_TR
Ukrainian	Ukraine	uk_UA
Vietnamese	Vietnam	vi_VN

Usually, both elements of a locale are important. For example, the names of weekdays and months, in English, for Canada and Great Britain are formatted identically, but dates are formatted differently. Therefore, specifying an English language locale is insufficient. Also, specifying only a country as the locale is insufficient. 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 time begins. Any date value and any time value prior to its epoch is invalid.

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.

LiveCycle Designer ES defines day one 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 can 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 that represent 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	25/07/85
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. The four general categories of date formats are short, medium, long, and full. The following table contains examples of different date formats from different locales for each of the categories.

Locale identifier and description	Date format (Category)	Example
en_GB English (United Kingdom)	DD/MM/YY (Short)	08/18/92 08/04/05
fr_CA French (Canada)	YY-MM-DD (Medium)	92-08-18

Locale identifier and description	Date format (Category)	Example
de_DE German (Germany)	D. MMMM YYYY (Long)	17. Juni 1989
fr_FR French (France)	EEEE, ' le ' D MMMM YYYY (Full)	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

The following table lists examples of date formats for different locales. Each locale identifies the format of times that conform to the standards of that nation.

Locale identifier and description	Time format (Category)	Example
en_GB English (United Kingdom)	HH:MM (Short)	14:13
fr_CA French (Canada)	HH:MM:SS (Medium)	12:15:50
de_DE German (Germany)	HH:MM:SS z (Long)	14:13:13 -0400
fr_FR French (France)	HH ' h ' MM Z (Full)	14 h 13 GMT-04:00

Date and time picture formats

The following symbols must be used to create date and time patterns for date/time fields. Certain date symbols are only used in Chinese, Japanese, and Korean locales. These symbols are also specified below.

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` can be specified as the display pattern.

Date symbol	Description	Formatted value for English (USA) locale where the locale-sensitive input value is 1/1/08 (which is January 1, 2008)
D	1 or 2 digit (1-31) day of the month	1
DD	Zero-padded 2 digit (01-31) day of the month	01
J	1, 2, or 3 digit (1-366) day of the year	1
JJJ	Zero-padded, three-digit (001-366) day of the year	001
M	One- or two-digit (1-12) month of the year	1
MM	Zero-padded, two-digit (01-12) month of the year	01
MMM	Abbreviated month name	Jan
MMMM	Full month name	January
E	One-digit (1-7) day of the week, where (1=Sunday)	3 (because January 1, 2008 is a Tuesday)
EEE	Abbreviated weekday name	Tue (because January 1, 2008 is a Tuesday)
EEEE	Full weekday name	Tuesday (because January 1, 2008 is a Tuesday)
YY	Two-digit year, where numbers less than 30 are considered to fall after the year 2000 and numbers 30 and higher are considered to occur before 2000. For example, 00=2000, 29=2029, 30=1930, and 99=1999	08
YYYY	Four-digit year	2008
G	Era name (BC or AD)	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	1
WW	Two-digit (01-53) ISO-8601 week of the year, where week 1 is the week containing January 4	01

Several additional date patterns are available for specifying date patterns in Chinese, Japanese, and Korean locales.

Japanese eras can be represented by several different symbols. The final four era symbols provide alternative symbols to represent Japanese eras.

CJK date symbol	Description
DDD	The locale's ideographic numeric valued day of the month
DDDD	The locale's tens rule ideographic numeric valued day of the month
YYY	The locale's ideographic numeric valued year
YYYYY	The locale's tens rule ideographic numeric valued year
g	The locale's alternate era name. For the current Japanese era, Heisei, this pattern displays the ASCII letter H (U+48)
gg	The locale's alternate era name. For the current Japanese era, this pattern displays the ideograph that is represented by the Unicode symbol (U+5E73)
ggg	The locale's alternate era name. For the current Japanese era, this pattern displays the ideographs that are represented by the Unicode symbols (U+5E73 U+6210)
g	The locale's alternate era name. For the current Japanese era, this pattern displays the full width letter H (U+FF28)
g g	The locale's alternate era name. For the current Japanese era, this pattern displays the ideograph that is represented by the Unicode symbol (U+337B)

Time symbol	Description	Locale-sensitive input value	Formatted value for English (USA) locale
h	One- or two-digit (1-12) hour of the day (AM/PM)	12:08 AM or 2:08 PM	12 or 2
hh	Zero-padded 2 digit (01-12) hour of the day (AM/PM)	12:08 AM or 2:08 PM	12 or 02
k	One- or two-digit (0-11) hour of the day (AM/PM)	12:08 AM or 2:08 PM	0 or 2
kk	Two-digit (00-11) hour of the day (AM/PM)	12:08 AM or 2:08 PM	00 or 02
H	One- or two-digit (0-23) hour of the day	12:08 AM or 2:08 PM	0 or 14
HH	Zero-padded, two-digit (00-23) hour of the day	12:08 AM or 2:08 PM	00 or 14
K	One- or two-digit (1-24) hour of the day	12:08 AM or 2:08 PM	24 or 14
KK	Zero-padded, two-digit (01-24) hour of the day	12:08 AM or 2:08 PM	24 or 14
M	One- or two-digit (0-59) minute of the hour **Note:** You must use this symbol with an hour symbol.	2:08 PM	8

Time symbol	Description	Locale-sensitive input value	Formatted value for English (USA) locale
MM	Zero-padded, two-digit (00-59) minute of the hour **Note:** You must use this symbol with an hour symbol.	2:08 PM	08
S	One- or two-digit (0-59) second of the minute **Note:** You must use this symbol with an hour and minute symbol.	2:08:09 PM	9
SS	Zero-padded, two-digit (00-59) second of the minute **Note:** You must use this symbol with an hour and minute symbol.	2:08:09 PM	09
FFF	Three- digit (000-999) thousandth of the second **Note:** You must use this symbol with an hour, minute, and seconds symbol.	2:08:09 PM	09
A	The part of the day that is from midnight to noon (AM) or from noon to midnight (PM)	2:08:09 PM	PM
z	ISO-8601 time-zone format (for example, Z, +0500, -0030, -01, +0100) **Note:** You must use this symbol with an hour symbol.	2:08:09 PM	-0400
zz	Alternative ISO-8601 time-zone format (for example, Z, +05:00, -00:30, -01, +01:00) **Note:** You must use this symbol with an hour symbol.	2:08:09 PM	-04:00
Z	Abbreviated time-zone name (for example, GMT, GMT+05:00, GMT-00:30, EST, PDT) **Note:** You must use this symbol with an hour symbol.	2:08:09 PM	EDT

Reserved symbols

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

Symbol	Description
?	When submitted, the symbol matches any one character. When merged for display, it becomes a space.
*	When submitted, the symbol matches 0 or Unicode white space characters. When merged for display, it becomes a space.
+	When submitted, the symbol matches one or more Unicode white space characters. When merged for display, it 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: • 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(1)`	`M/D/YY` (if en_US locale is set)
`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 date format style as follows: • 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-MM-jj`
`LocalDateFmt(3, "de_CH")`	`t. MMMM jjjj`
`LocalDateFmt(4, "fr_FR")`	`EEEE j MMMM 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: • 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.

Expression	Returns
`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: • 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(1)`	`h:MM A` (if en_US locale is set)
`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 9.

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 9.

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 9.

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 9.

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 9.

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 9.

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.

Parameter	Description
`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 9.

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.

Parameter	Description
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 9.

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 9.

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 9.

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.
`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`

Expression	Returns
`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 a reference syntax to an existing object.

Syntax
`Exists(`*v*`)`

Parameters

Parameter	Description
`v`	A valid reference syntax expression. If `v` is not a reference syntax, 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 a reference syntax.
`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 a reference syntax with a non-null, non-empty, or non-blank value.

Syntax
`HasValue(`*v*`)`

Parameters

Parameter	Description
`v`	A valid reference syntax expression. If `v` is not a reference syntax, 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 79.

Ref

Returns a reference to an existing object.

Definition
`Ref(v)`

Parameters

Parameters	Description
`v`	A valid string representing a reference syntax, property, 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 84.

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`

Expression	Returns
`Concat("Tony", Space(1), "Blue")`	`Tony Blue` See also "Space" on page 87.
`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 92.

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 date or time format. 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 87.

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 88.
`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 87.
`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"`

Expression	Returns
`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 87.

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 37.
`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 37.

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 83.
`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.

Note: The `Get` function only runs if a form is certified. Adobe Acrobat® and Adobe Reader® cannot verify that the form is certified until after the `initialize` event initiates. To use the `Get` function on certified forms prior to the form rendering, use the `docReady` event.

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 an SQL query to the specified website.

Post

Posts the given data to the specified URL.

Note: The `Post` function only runs if a form is certified. Acrobat and Adobe Reader cannot verify that the form is certified until after the `initialize` event initiates. To use the `Post` function on certified forms prior to the form rendering, use the `docReady` event.

Syntax

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

Parameters

Parameter	Description
s1	The URL to post to.
s2	The data to post. If the function cannot post the data, it returns an error.
s3 (Optional)	A string containing the content type of the data to post. Here are valid content types: • 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 ensures 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. Here are valid code page names: • 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 ensures 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.

Note: The `Put` function only runs if a form is certified. Acrobat and Adobe Reader cannot verify that the form is certified until after the `initialize` event initiates. To use the `Put` function on certified forms prior to the form rendering, use the `docReady` event.

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. Here are valid code page names: • 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 ensures 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 `pub/fubu.xml` file. Otherwise, this function returns an error.

Index

W