Loan Custom Field Calculations

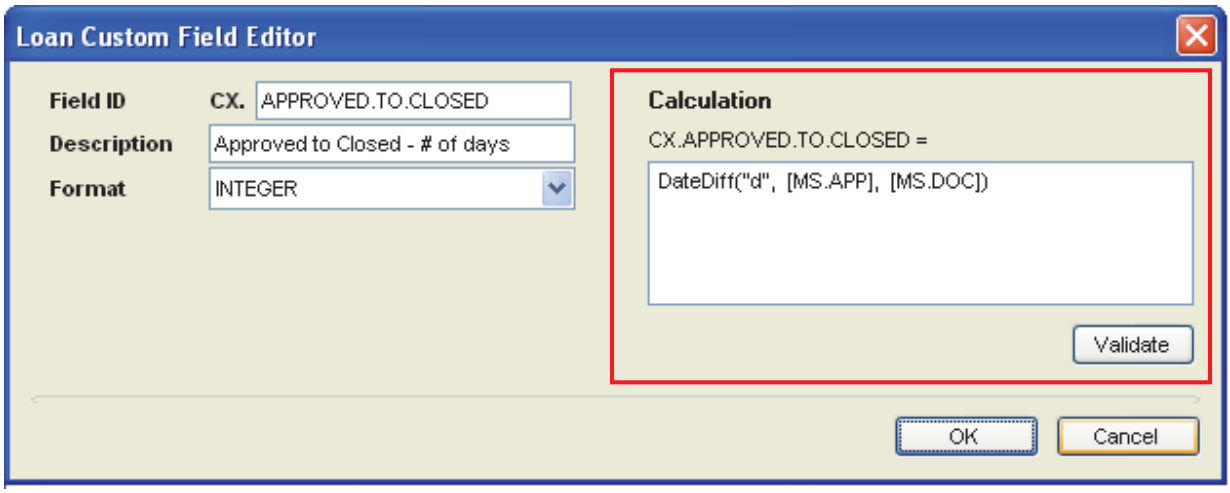
For use with Encompass Banker Edition

The Loan Custom Fields tool provides the ability to create calculations for both pre-defined and user-defined custom field IDs. A custom calculation is an expression that returns a number or text value, which is then saved into the associated custom field. This document describes the types of operations you can use in a custom field calculation, and provides examples of how to use each one.

NOTE: *Loan custom field calculations can also be used in advanced coding for business rules. See the Advanced Coding for Business Rules document.*

To Define a Custom Field with a Calculation:

1. On the menu bar, click **Encompass**, and then click **Settings**.
2. On the left panel, click **Loan Setup**, and then click **Loan Custom Fields**.
3. Double-click to open a pre-defined custom Field (such as CUST01FV).
   - To create a user-defined custom field ID, click the **New** icon and type the Field ID.
4. Enter a Description, and then select a Format for the field. Refer to the online help for more information regarding the field formats.
5. Enter the calculation.
6. To validate the calculation syntax, click the **Validate** button and then click **OK** to the message.
7. When finished, click **OK**.



Using Loan Field Values

Most custom calculations need to use the value of loan fields in order to calculate the value for the associated custom field. You can insert a reference to any loan field which has a defined field ID by placing the field's ID within square-brackets ([]). For example, [136] references the purchase price. Any of the more than 7000 standard fields shipped as part of Encompass, or the loan custom fields you have already defined, can be referenced in this manner.

By default, when referencing a field, the type of value returned (numeric or text) is based on the type of the underlying field. For example:

- [1109] (loan amount) returns a numeric value
- [4000] (borrower first name) returns a string value

There is one exception to this rule: when a numeric field has not yet been populated, a reference to that field will return an empty string. As a result, you must use care when writing calculations to consider what will happen when the fields used in the calculation are empty (refer to “Calculation Errors” on page 9).

November, 2015

Operations and Features

A custom calculation can contain any combination of the following operations and features:

- Simple arithmetic operations, such as addition and subtraction
- Mathematical operations, such as exponentiation and absolute value
- Text-based operations, such as concatenation, truncation, and extraction of characters
- Date-based operations, such as adding a fixed number of days or months to a date
- Values of other fields from the same loan
- Branching and logical operations, such as if...then, AND, and OR

Arithmetic Operations

Arithmetic operations form the basis for most custom calculations and consist of addition (+), subtraction (-), multiplication (*), and division (/).

For example, the following custom calculation returns 2% of the value stored in field ID 1109 (loan amount).

```
0.02 * [1109]
```

Encompass custom calculations use standard arithmetic order-of-operations rules and allow for use of parentheses to force a specific execution order.

```
0.02 * (([1109] + [44])
```

An important consideration when authoring custom calculations is whether or not the specified arithmetic operations can be performed based on the values in the fields. In the example above, the calculation assumes that both fields 1109 and 44 contain valid numeric data. If either field is blank or contains non-numeric data, the calculation will fail and the custom field's value will be cleared (refer to “Calculation Errors” on page 9).

Safe Operations

When you are unsure of the potential validity of an arithmetic expression within a calculation, you can use one of the following built-in functions in place of the standard operator.

Function	Description	Example
Sum(x, y, z, ...)	Adds the specified values. If any operand cannot be converted to a number, the entire expression returns an empty value.	Sum([1109], [44])
SumAny(x, y, z, ...)	Adds the specified values, ignoring those that cannot be converted to numbers. If none of the values can be converted to a number, the function returns a blank value.	SumAny([1109], [44])
Diff(x, y)	Evaluates to x – y if both values are numeric. If either value is non-numeric, the function returns a blank value.	Diff([1109], [44])
Mult(x, y, z, ...)	Returns the product of the operands. If any of the operands cannot be converted to a number, the function returns a blank value.	Mult([1109], [44])
MultAny(x, y, z, ...)	Returns the product of all operands which can be converted to numbers. If none of the values can be converted to a number, returns a blank value.	MultAny([1109], [44])
Div(x,y)	Returns the value of x / y if both operands can be converted to numeric values. Otherwise, returns a blank value.	Div([1109], [44])

Using these operators, it is possible to write a "safe" version.

Example:

```
0.02 * ([1109] + [44])
```

Rewritten using the safe operators:

```
Mult(0.02, Sum([1109], [44]))
```

If either field 1109 or field 44 is empty or non-numeric, this expression will evaluate to an empty value. Expressions of this form are most commonly needed when using the branching function IIF. Refer to “Branching and Logic Operations” on page 6.

Mathematic Operations

Within custom calculations you can employ any of a number of pre-defined mathematical functions to calculate the resulting value. The following table lists the supported functions.

Function	Description	Example
Abs(x)	Returns the absolute value of the specified argument.	Abs([101] - [102])
Min(x, y, z, ...)	Returns the smallest of a set of values.	Min([102], [103], [104])
Max(x, y, z, ...)	Returns the largest of a set of values.	Max([102], [103], [104])
LMedian(x, y, z, …)	Returns the median value of a set of numbers. If an even number of values is specified, the lower of the two middle values will be returned.	LMedian([#67], [#1450], [#1414])
UMedian(x, y, z, …)	Returns the median value of a set of numbers. If an even number of values is specified, the higher of the two middle values will be returned.	UMedian([#67], [#1450], [#1414])
Sqrt(x)	Returns the square root of a value.	Sqrt(0.02 * [1109])
Log(x)	Returns the natural logarithm (base-*e*) of the value.	Log(1000 + [910])
Log10(x)	Returns the base-10 logarithm of the value.	Log10(1000 + [910])
Exp(x)	Returns the value of e^x.	Exp([1171] / 100)
Pow(x, y)	Returns the value of x^y.	Pow([1171], 5)
Sgn(x)	Returns 1 if x > 0, 0 if x = 0, or -1 if x < 0.	Sgn([1093])
Round(x, precision)	Rounds the value x to a number of decimal places specified by *precision*.	Round([1109], 2)
Trunc(x, precision)	Truncates the value x to a number of decimal places specified by *precision*.	Trunc([1109], 2)
XInt(x, *default*)	Converts the value x to an integer. If x is a string, the integer value it represents is returned. Non-integral values are rounded to the nearest integer. If the value cannot be converted successfully, the optional *default* value is returned. If no default is provided, the value 0 is returned.	XInt(“230”) or XInt([1109], 1)
XDec(x, *default*)	Converts the value x to a decimal. If x is a string, the numeric value it represents is returned. If the value cannot be converted successfully, the optional *default* value is returned. If no default is provided, the value 0 is returned.	XDec(“245.112”) or XDec([1109], -1)

Text-Based Operations

Depending on the type of field with which the calculation is associated, your calculation may need to return a text-based value instead of a numeric value. For example, if your field type is defined to be "Y/N", then your custom calculation should evaluate to either "Y" or "N", since these are the only two valid values for this field type.

The most basic string operation that can be performed is concatenation, which is done by using the ampersand (&) and operator. For example:

Builds the borrower's full name from their first and last names with a space separating them.

```
[4000] & " " & [4002]
```

Returns the value of field ID 1109 (loan amount) preceded by the dollar sign and followed by a space and USD, such as $200,000.00 USD.

```
"$" & [+1109] & " USD"
```

Note that literal strings, such as " " (space), "$", and the " USD" are denoted by enclosing the text in double quotes. If the need arises to include the double-quote character itself in a string literal, it should occur twice.

For example, assuming that field 4000 has the value "Joe", the expression:

```
[4000] & " is the ""primary"" borrower for the loan."
```

would evaluate to:

```
Joe is the "primary" borrower for the loan.
```

In addition to concatenation, Encompass provides the following string-based functions to assist you in authoring custom calculations.

Function	Description	Example
Trim(x)	Removes any white space characters from the beginning and end of the value x.	Trim([4000] & " " & [4002])
Left(x, n)	Returns the left-most n characters from the value x. If x is shorter than n characters, the whole value is returned.	Left([4002], 5)
Right(x, n)	Returns the right-most n characters from the value x. If x is shorter than n characters, the whole value is returned.	Right([4002], 5)
Mid(x, start, *length*)	Returns the substring of the value x that starts at the specified locations and has the specified length. If *length* is omitted, the entire string after *start* is returned. The start position is 0-based.	Mid([4002], 3, 2)
InStr(x, y)	Returns the first position of the substring y within the string x. The comparison is case-sensitive.	InStr([4002], "mith")
Int2Text(x)	Converts an integer value to its spelled out representation, e.g. “Five Hundred Thirty-Four.”	Int2Text([4])
Dec2Text(x)	Converts a decimal value to its spelled out representation, e.g. “Sixty-Five and Fifty-Three Hundredths.”	Dec2Text([3])
Money2Text(x)	Converts a decimal value to its spelled out representation using ‘dollars’ and ‘cents’ notation, e.g. “Sixty-Five Dollars and Fifty-ThreeCents.”	Money2Text([1109])
LCase(x)	Returns the value of x with all letters converted to lower case.	LCase([4002])
UCase(x)	Returns the value of x with all letters converted to upper case.	UCase([4002])
Replace(x, y, z)	Replaces all instances of the substring y within the string x with the replacement value z, and returns the resulting string.	Replace([1264], "Investor", "Lender")

Date-Based Operations

Just as a custom calculation can yield a numeric or text result, it can also yield a date result when the underlying field type is of type DATE. Note that fields that have the MONTHDAY format do not contain full date values and cannot be used with the functions below.

Function	Description	Example
Day(x)	Returns the day portion of the date value x. The parameter x must be a valid date or the calculation will fail.	Day([1402])
Month(x)	Returns the month portion of the date value x. The parameter x must be a valid date or the calculation will fail.	Month([1402])
Year(x)	Returns the year portion the of date value x. The parameter x must be a valid date or the calculation will fail.	Year([1402])
DateAdd(period, count, x)	Add a fixed number of days, months, or years to the date value x. The period should be one of "d" (days), "m" (months), or "yyyy" (years). The count is the number of days/months/years to add. If any parameter is invalid, the entire calculation will result in a blank value.	DateAdd("yyyy", 1, [1402])
XDateAdd(period, count, x)	Add a fixed number of days, months, or years to the date value x. The period should be one of "d" (days), "m" (months), or "yyyy" (years). The count is the number of days/months/years to add. If any parameter is invalid, an empty value is returned from the function and the calculation will proceed.	XDateAdd("yyyy", 1, [1402])
DateDiff(period, x, y)	Computes the difference between the dates x and y in days, months or years. The period should be one of "d" (days), "m" (months) or "yyyy" (years). If any parameter is invalid, the entire calculation will result in a blank value.	DateDiff("d", [1402], [1403])
XDateDiff(period, x, y)	Computes the difference between the dates x and y in days, months or years. The period should be one of "d" (days), "m" (months) or "yyyy" (years). If any parameter is invalid, an empty value is returned from the function and the calculation will proceed.	XDateDiff("d", [1402], [1403])
XDate(x, *default*)	Converts the value x to a date. If x is a string, the date value it represents is returned. If the value cannot be converted successfully, the optional *default* value is returned. If no default is provided, the date 1/1/1 is returned.	XDate([1402], "11/30/2010")
XMonthDay(x, *default*)	Converts the value x to a “month-day” value. A month-day is represented as a date within the year 2010. For example, XMonthDay(“3/15”) would return the date 3/15/2010. If the value cannot be converted to a month-day value, the optional default is provided.	XMonthDay(“3/15”)
Today	Returns today’s date.	DateAdd("d", 7, Today)

Calendar-Based Operations

If your custom field calculation requires access to your company's compliance calendar, the functions below provide the necessary operations to add or adjust dates based on either the Postal or Business calendar.

Function	Description	Example
Calendar.AddBusinessDays (date, count, moveToNext)	Adds the specified number of days from your company's Business Calendar to the date provided. The moveToNext parameter is a Boolean which indicates if the date should first be advanced to the next business day if the date specified is not a business day.	Calendar.AddBusinessDays ([763], 5, true)
Calendar.AddPostalDays (date, count, moveToNext)	Adds the specified number of days from the US Postal Calendar to the date provided. The moveToNext parameter is a Boolean which indicates if the date should first be advanced to the next business day if the date specified is not a business day.	Calendar.AddPostalDays ([763], 5, true)
Calendar.AddRegZBusinessDays (date, count, moveToNext)	Adds the specified number of days from the Reg-Z Business Day Calendar to the date provided. The moveToNext parameter is a Boolean which indicates if the date should first be advanced to the next business day if the date specified is not a business day.	Calendar.AddRegZBusinessDays ([763], 5, true)

Branching and Logic Operations

Many custom calculations require complex branching (if…then) logic in order to arrive at the desired value. To accommodate this need, the custom calculations provide the IIF() function, which can be used to express if…then…else logic. The basic syntax of this function is as follows:

```
IIF(Boolean expression, True value, False value)
```

The first argument to IIF() is an expression that evaluates to either TRUE or FALSE. If that expression evaluates as TRUE, then the IIF function returns the value specified in the second parameter (the "True value"). Otherwise, the IIF function returns the third parameter, the "False value."

For example, the following custom calculation returns 2% of the loan amount for loan amounts above $100,000 and 5% of the loan amount for loans below $100,000:

```
IIF([1109] > 100000, 0.02 * [1109], 0.05 * [1109])
```

The Boolean expression of an IIF statement can use the AND and OR operations to perform more complex logic, for example:

```
IIF([1109] > 100000 AND [1335] < 20000, 0.02 * [1109], 0.05 * [1109])
```

Often, you may encounter the need to handle more than just two cases (a TRUE case and a FALSE case). In these scenarios, you can use nested IIF statements. For example, the following code demonstrates performing a calculation based on the purpose of the loan.

```
IIF([1811] = "PrimaryResidence", 0.05 * [1109],
     IIF([1811] = "SecondHome", 0.02 * [1109],
     IIF([1811] = "Investor", 0.01 * [1109], 0)))
```

The expression above evaluates to 5% of the loan amount if the loan is for the borrower's primary residence, 2% if it's a second home, 1% for an investment property and return the value "0" if the loan purpose has not yet been specified.

IMPORTANT: *When the IIF function is invoked, the calculation engine first evaluates both the second and third parameters of the function before evaluating the Boolean expression. Therefore, both parameters need to evaluate to valid values even though only one value will be used. For example, the following IIF statement will fail.*

```
IIF(5 > 0, 1, 2 * [4000])
```

The expression 2 * [4000] is typically invalid since field 4000 contains the borrower's first name (which is typically not numeric). Even though the logical expression 5 > 0 will always evaluate to TRUE (and thus the value returned by IIF will always be 1), the failure of the third parameter to evaluate successfully will cause the entire calculation to fail.

To work around this issue, you can use any of the following techniques:

- Use the safe arithmetic operations, e.g. IIF(5 > 0, 1, Mult(2, [4000])). These operations ensure that no errors will occur.
- Use the field modifiers, e.g. IIF(5 > 0, 1, 2 * [#4000]). The numeric conversion modifier will convert non-numeric values to 0, allowing the calculation to be carried out successfully (refer to “Calculation Errors” on page 9).
- Use a combination of two custom fields, for example:

  ```
  CX.FIELD1 = 2 * [4000]
  CX.FIELD2 = IIF(5 > 0, 1, [CX.FIELD1])
  ```

  When field 4000 is non-numeric, the calculation for field CX.Field1 will be invalid and, as a result, the field will be blank. However, the expression for CX.Field2 can now be evaluated successfully since the substitution of the value CX.FIELD1 will always work (even if the value happens to be blank).

As a second example, the expression above that branches based on the loan purpose would be rewritten as follows to ensure safe evaluation regardless of the value in field 1109:

```
IIF([1811] = "PrimaryResidence", 0.05 * [#1109],
     IIF([1811] = "SecondHome", 0.02 * [#1109],
     IIF([1811] = "Investor", 0.01 * [#1109], 0)))
```

In addition, the custom calculation engine provides a few functions which can be used to determine the state of a field so you can add condition logic that behaves appropriately.

Function	Description	Example
IIF(x, truepart, falsepart)	Returns the truepart if x is true, the falsepart otherwise.	IIF([#1109] > 100000, 10, 20)
IsEmpty(x)	Returns a boolean indicating if the value x is the empty string.	IsEmpty([1109])
IfEmpty(x, val)	Returns the value *x* if it is non-empty, otherwise returns the value *val*.	IfEmpty([1109], 0)
IsNumeric(x)	Returns a boolean indicating if the value x can be converted into a numeric value.	IsNumeric([1109])
IsDate(x)	Returns a boolean indicating if the value x can be converted to a valid date.	IsDate([1402])

List-Based Operations

Evaluating a custom calculation will frequently require logic which involves looking up a value in list of possible values or within a range of values. These operations could be carried out using one or more nested IIF() expressions as demonstrated above, but the calculation engine offers several functions to simplify this task.

Function	Description	Example
Match(x, value0, value1, …)	Returns the index of the first value in the list that matches x. If not match is found, the value -1 is returned.	Match([608], "Fixed", "GraduatedPaymentMortgage", "AdjustableRate", "OtherAmortizationType")
Range(x, value0, value1, …)	Returns the index of the first value which is greater than x.	Range([1109], 100000, 200000, 500000)
RangeLow(x, value0, value1, …)	Returns the index of the first value which is greater than or equal to x.	RangeLow([1109], 100000, 200000, 500000)
Pick(x, value0, value1, …)	Returns the value whose index is x.	Pick([#16] - 1, "1 unit", "2 units", "3 units", "4 units")
Count(value0, value1, …)	Returns the number of parameters that are non-empty.	Count([4000], [4002], [98], [99])

Using these functions, this calculation:

```
IIF([1811] = "PrimaryResidence", 0.05 * [#1109],
     IIF([1811] = "SecondHome", 0.02 * [#1109],
     IIF([1811] = "Investor", 0.01 * [#1109], 0)))
```

could be rewritten as:

```
Pick(Match([1811], "PrimaryResidence", "SecondHome", "Investor", ""),
     0.05 * [#1109], 0.02 * [#1109], 0.01 * [#1109], 0)
```

Advanced Functions

The Encompass Custom Calculation Engine leverages the Visual Basic.NET programming language when evaluating your calculation. As a result, you may make use of any function which is provided as part of the VB.NET programming language. For a complete reference of these functions, visit Microsoft's VB.NET Reference.

Keep in mind that your calculation should not invoke any method which can trigger the display of a user interface of any kind as this can cause Encompass or the Encompass Server to fail.

Calculation Errors

There are two types of errors that can occur when authoring custom field calculations:

- Syntax errors, which are detected when you validate or save the custom calculation
- Runtime errors, which occur during the evaluation of the calculation which Encompass is running

When a runtime error occurs during the evaluation of a custom field calculation, Encompass will clear the field. For example, consider the field calculation:

```
[1109] * 2
```

If field 1109 is blank, this calculation will fail and the custom field will also be blank. Referencing an unpopulated numeric field, returns an empty string. When field 1109 takes on a numeric value, the calculation will succeed and the custom field's value will be populated appropriately.

In general, if an error occurs in any part of the calculation, the entire calculation will fail. To avoid this, use the build-in functions (Sum(), Diff(), etc.) or use the field modifiers described below. Either technique can be used to avoid errors that might otherwise short circuit your calculation.

Consider the following expression:

```
[101] + [102] + [103] + [104] + [105]
```

If all five referenced fields contain numeric data, this expression will evaluate to the sum of these values and the result saved into the custom field. However, if any one or more of these fields is unpopulated, the expression will fail since a string is not valid within a summation. As a result, the custom field's value will be cleared.

There are two possible resolutions to this issue:

1 Use the "safe" arithmetic operators. The expression above could be rewritten as:

```
SumAny([101], [102], [103], [104], [105])
```

By definition, the SumAny() function ignores any values which are non-numeric and returns the sum of those that are. If none of the values are numeric, the SumAny() function returns an empty string.

2 Use the numeric field modifier. Field modifiers allow you to provide additional instructions on the format in which the field should be inserted into the code. Modifiers are inserted within the square brackets but before the field ID, such as [#101]. The following modifiers are available.

Modifier	Description	Example
#	Forces a numeric value to be returned. If the field cannot be converted to a number, the value 0 is returned.	[#1109]
-	Forces the field value to be returned as an unformatted string. For example, a typical numeric value might return "200000.00".	[-1109]
+	Forces the field value to be returned as a formatted string. For example, a typical numeric value might return "200,000.00".	[+1109]
@	Forces the field value to be returned as a date. If the field cannot be converted to a date, the date 1/1/1 is returned.	[@1109]

Using the "#" modifier, the expression above could be rewritten as:

```
[#101] + [#102] + [#103] + [#104] + [#105]
```

Note that unlike the SumAny() function that will return a blank value if all of the referenced fields are empty, this expression would evaluate to 0 in that scenario (since each operand will evaluate to 0).

When you place a reference to another loan field within a custom field calculation, that field will automatically be updated whenever the value of the referenced field is modified. As a result, you must avoid circular dependencies in your custom field calculations. For example, say you define the following custom field calculations:

```
CX.FIELD1 = [1109] + [CX.FIELD2]
CX.FIELD2 = [CX.FIELD3] * 2
CX.FIELD3 = [CX.FIELD1] + 20000
```

Individually, each calculation is valid, but a circular dependency exists between the three fields. The Custom Field Editor will notify you of circular dependencies when they exist.