OpenText™ Exstream™

Using Logic to Drive an Application

Design and Production Documentation

Release 16.6.0

OpenText™ Exstream
Using Logic to Drive an Application
Rev.: 2019-Apr-30

This documentation has been created for software version 16.6.0.

It is also valid for subsequent software versions as long as no new document version is shipped with the product or is published at https://knowledge.opentext.com.

Open Text Corporation
275 Frank Tompa Drive, Waterloo, Ontario, Canada, N2L 0A1

Tel: +1-519-888-7111
Toll Free Canada/USA: 1-800-499-6544 International: +800-4996-5440
Fax: +1-519-888-0677

Support: https://support.opentext.com
For more information, visit https://www.opentext.com

Copyright © 2019 Open Text. All rights reserved.

Trademarks owned by Open Text.

One or more patents may cover this product. For more information, please visit, https://www.opentext.com/patents

Disclaimer

No Warranties and Limitation of Liability

Every effort has been made to ensure the accuracy of the features and techniques presented in this publication. However, Open Text Corporation and its affiliates accept no responsibility and offer no warranty whether expressed or implied, for the accuracy of this publication.

Contents

Chapter 1: About Logic

In Exstream, logic lets you use customer data to personalize your applications. By using logic in the form of rules, formulas, and functions in your application, you can connect customer data with your application and design objects to control the appearance of customer output, perform customized calculations, or manipulate customer data for use with an application. This connection between data and design lets you define how the engine applies data and control objects by timing when or if they are placed into customer output. Logic can be used to control a wide variety of actions, from creating business logic that manipulates data to controlling the inclusion or exclusion of objects in your application. Since each type of logic (rules, formulas, and functions) has specific uses, you also have the option to combine a specific logic element (rules, formulas, or functions) with any other logic element to accomplish an even wider variety of actions.

For more information about how each of the individual types of logic can be used, see "Types of Logic" below.

Logic in Exstream can be created using several different methods, including several methods that do not require programming experience. The coding syntax used to program logic in Exstream is similar to programming in Visual Basic; however, there are some key differences, which are explained in the last chapter of this guide. If you intend to code logic for use with Exstream, you should be familiar with programming in Visual Basic.

This guide describes what rules, formulas, and functions are and the design tools you use to create them.

1.1 Types of Logic

Depending on the actions you want to perform using logic, you will use a different object in Exstream. Use the following table to help you decide what type of logic to use with your application.

Types of logic

Use this	To do this
Rules	Include or exclude an object from final output based on customer information. For example, when a particular customer meets all conditions in the logic, you can set the rule to include a special message or specific document.

Types of logic, continued

Use this	To do this
Formulas	Depending on how you build the logic for a formula, a formula can be used in the following ways: • Calculate values using information available in your database. The most common use of a formula is to search through arrays. For example, you can create a formula that calculates the total amount a customer owes by adding all the charges a customer has made for the month. • Manipulate data for use with a local variable. For example, by adding the `Left`, `Mid`, or `Right`built-in functions into a formula, you can break a large chunk of data into smaller, more usable parts, and then use the formula to define those parts as local variables you can either insert into a customer document or reuse in other logic.
Library functions	Depending on how you build the logic for a Library function, a Library function can be used in the following ways: • Cause an action to occur based on information in the customer data. For example, you can use a function to look up or write a specific reference file if a customer has a certain type of account. • Search data for specific types of information. For example, you can use a function to return a customer's anniversary month only, even if the month, day, and year are listed in the data.
Built-in functions	Perform common or general tasks on the data. Built-in functions are preset functions that only require you to insert specified data to perform the assigned action of the built-in function. For example, if you need to calculate a mathematical value, such as an average or the absolute value of a number, you can use the pre-programed logic of a built-in function rather than building the code from scratch.

Chapter 2: Rules

Rules allow you to set objects to be included or excluded based on the customer data. For example, suppose you want to include an object only if the customer is from a specific country/region. You can connect a rule to the customer information that specifies the country/region in which a customer lives. When the design is produced as output, the rule causes the engine to search through the customer information to verify whether the customer is from the correct country/region. If the customer data indicates that the customer is from the correct country/region, the object is included in the output. Otherwise the object is not included in the customer output.

To work with rules in Exstream, you must complete the following tasks:

1. Complete one of the following tasks:

To	Complete this task
Create a rule for a specific object	"Accessing the Rule Tab" on the next page
Create a rule with logic that is limited to use with a single object	"Creating an Unnamed Rule" on page 14
Create a reusable rule that contains logic you can apply to multiple objects	"Creating a Library Rule" on page 14
Create a rule to control the priority order in which campaigns are selected for inclusion in customer output. This type of rule is used only with campaigns.	"Creating a Priority Rule" on page 18

2. "Building the Logic for a Rule" on page 19
3. "Setting Rule Timing for the Engine" on page 28

You can also complete the following optional task as needed:

- "Testing or Troubleshooting Rules" on page 30

If you are adding rules to objects in Designer, you can also complete the following optional task as needed:

- "Reviewing the Rules Applied to Your Design" on page 32

If you are adding rules to objects in Designer, you also have the option to run a design rules report to review the business logic you add to your design. See "Reviewing the Rules Applied to Your Design" on page 32.

2.1 Accessing the Rule Tab

All rules are set up from the **Rule** tab. However, since rules are generally stored either with the object to which they apply or in the Library, you can encounter the **Rule** tab in multiple locations across Exstream. Use one of the following procedures to access the **Rule** tab, when needed, for rules applied to objects, Library rules, or priority rules.

To access the **Rule** tab, do one of the following sequences of steps:

For this object	Do this
Data files	1. In Design Manager, from the Library, drag the data file to the Property Panel. 2. Click the **Advanced** tab. 3. Click in the **Rule** box. The **Rule** dialog box opens. 4. Click the **Rule** tab.
Documents, pages, paragraph objects, section objects, messages, and campaigns	1. In Design Manager, from the Library, drag the object to the Property Panel. 2. Click the **Targeting** tab. 3. Click in the **Rule** box. The **Rule** dialog box opens. 4. Click the **Rule** tab.
Library rules	1. In Design Manager, from the Library, drag the Library rule to the Property Panel. 2. Click in the **Logic** area. The **Rule** dialog box opens. 3. Click the **Rule** tab.
Priority rules on campaigns	1. In Design Manager, from the Library, drag the campaign to the Property Panel. 2. Click the **Priority** tab. 3. From the **Method for specifying priority of this campaign compared to other campaigns** drop-down list, select **Priority is set by a rule for each customer**. 4. Click in the **Rule** box. The **Rule** dialog box opens. 5. Click the **Rule** tab.

For this object	Do this
Text or embedded objects **Note:** You can apply text rules only to text placed in text boxes, messages, paragraphs, and tables.	1. In Designer, from your design, select the object to make it active. 2. Highlight the necessary content: - For text, highlight the text as needed. Be sure to highlight all the necessary text, including spaces. Text on which a rule has previously been applied appears with a pink wavy underline. - For embedded objects, select the anchor character. 3. Right-click the highlighted text, and, from the shortcut menu, select either **Add Text Rule** or **Edit Text Rule**. 4. Click the **Rule** tab. **Tip:** Text rules are recommended only if you are controlling a substantial amount of text or an embedded object by rule. To control other amounts of text, consider whether using regular variables or using a string variable with a string table would provide the results you need.
Table rows or columns	1. In Designer, select the table row or column to make it active. 2. On the row or column you want to control, click ◆. If you are controlling a group of columns or rows, click ◆ on the first row or column in the group 3. From the short-cut menu, select **Properties**. The **[Object] Properties** dialog box opens. 4. Click the **Rule** tab.
All other objects in Designer	1. In Designer, from your design, select the object to make it active. 2. Click ▤. The **[Object] Properties** dialog box opens. 3. Click the **Rule** tab.

2.2 Unnamed Rules

Unnamed rules let you use logic to control the inclusion or exclusion of a single object. The logic for unnamed rules is stored directly with the object that it controls. For example, suppose you want to control the inclusion of a single page within a document. The unnamed rule would be created and stored in Design Manager with the page.

2.2.1 Creating an Unnamed Rule

1. Access the **Rule** tab for the object to which you want to apply the unnamed rule.

   For more information about accessing the **Rule** tab, see “Accessing the Rule Tab” on page 12.

2. Create the rule logic.

   For more information about creating rule logic, see “Building the Logic for a Rule” on page 19.

3. Click **OK**.

   The dialog box closes, and the unnamed rule is applied to the object.

4. From the Standard toolbar, click 💾.

   The object and the unnamed rule are saved.

2.3 Library Rules

Library rules let you create and store rule logic in the Design Manager Library and apply that logic to multiple objects. You can reference a Library rule anywhere you can create an unnamed rule. If you change the logic of the Library rule, the logic is automatically updated wherever the Library rule is referenced. This makes it easier to make changes to logic that is shared across multiple objects. For example, suppose you have text boxes, tables, and charts on a page. If all objects should only be applied to the customer output when the customer has a positive balance, and no other logic is needed to control these objects, you might want to apply a single Library rule to the objects.

To use a Library rule, you must complete the following tasks:

1. “Creating a Library Rule” below

2. “Referencing a Library Rule from an Object” on the next page

2.3.1 Creating a Library Rule

1. In Design Manager, from the Library, right-click the **Rules** heading, and, from the shortcut menu, select **New Rule**.

   The **New Rule** dialog box opens.

2. In the **Name** box, enter a name. In the **Description** box, enter a description (optional).

3. Click **Finish**.

   The Library rule opens in the Property Panel for you to define.

4. Click in the **Logic** area.

   The **Rule** dialog box opens.

5. Create the rule logic.

   For more information about creating rule logic, see “Building the Logic for a Rule” on page 19.

6. Click **OK**.

   The **Rule** dialog box closes and the logic is applied to the Library rule.

7. From the Standard toolbar, click 💾.

   The Library rule is saved.

2.3.2 Referencing a Library Rule from an Object

To apply a Library rule to an object, you must reference the Library rule. Keep in mind that applying a Library rule resets any code trace or watch level settings previously set for the object.

For more information about code trace and watch levels, see “Testing or Troubleshooting Rules” on page 30

To reference a Library rule from an object:

1. Access the **Rule** tab for the object to which you want to apply the Library rule.

   For more information about accessing the **Rule** tab, see “Accessing the Rule Tab” on page 12.

2. If the advanced options do not appear on the **Rule** tab, click + to toggle the appearance of advanced options.

3. In the **Library rule** area, click ± .

   The **Select Rule** dialog box opens.

4. Select the Library rule you want to apply.

5. Click **OK**.

   The **Select Rule** dialog box closes, and the name and folder of the Library rule appear in the **Library rule** area.

6. Click **OK**.

   The Library rule is applied to the object.

7. From the Standard toolbar, click 💾.

   The campaign and the priority rule are saved.

Note: After the rule is applied, the **Rule** tab is returned to the logic design view in which the rule was created. For example, if you add the rule while in code view and the Library rule was created in selection rule view, then the **Rule** tab returns to selection rule view.

For more information about rule logic design views, see "Logic Design Views in Which You Can Build Logic" on page 19.

2.3.3 Removing a Library Rule Reference

If you no longer want to reference a Library rule on an object, you can remove the reference to the Library rule from the object. Keep in mind that removing a Library rule reference also resets any code trace or watch level settings for this object.

For more information about code trace and watch levels, see "Testing or Troubleshooting Rules" on page 30.

1. Access the **Rule** tab for the object to which you want to apply the Library rule.

   For more information about accessing the **Rule** tab, see "Accessing the Rule Tab" on page 12.

2. If the advanced options do not appear on the **Rule** tab, click + to toggle the appearance of advanced options.

3. In the **Library rule** area, click ✖.

   The Library rule is removed from the object.

2.3.4 Converting an Unnamed Rule into a Library Rule

As you work with objects, there are multiple locations where you will be able to add rules. If you create a rule and decide that it would be useful as a Library rule, you can use the existing rule to create a Library rule. You can convert an unnamed rule into a Library rule whether the rule was created in Design Manager or in Designer.

Unnamed rules cannot be reused in other objects and do not appear in the **Rules** heading of the Library. However, Library rules (named rules) appear in the **Rules** heading in the Library and can be applied to numerous objects.

Note: When working with Library rules, keep in mind that the SBCS engine cannot process rules whose names contain DBCS characters. Rules that are used in both SBCS and DBCS applications must have names that are made up of SBCS characters only.

Additionally, the SBCS engine cannot correctly process rules that reference the following:

- Variables whose names contain DBCS characters
- Variables that contain DBCS content
- DBCS string literals

To create a Library rule from an existing rule:

1. Access the **Rule** tab for the object to which you want to apply the Library rule.

   For more information about accessing the **Rule** tab, see "Accessing the Rule Tab" on page 12.

2. If the advanced options do not appear on the **Rule** tab, click + to toggle the appearance of advanced options.

3. In the **Library rule** area, click ✚ .

   The **Folders** dialog box opens.

4. Select the folder where you want to create the Library rule.

5. Click **OK**.

   The **Folders** dialog box closes and the **Specify Library Rule Name** dialog box opens.

6. In the **Name** box, enter a name. In the **Description** box enter a description (optional).

7. Click **Finish**.

   The **Specify Library Rule Name** dialog box closes and the new Library rule appears in the **Library Rule** area.

2.3.5 Converting a Library Rule into an Unnamed Rule

As you work with objects, there are multiple locations where you will be able to add rules. If you have applied a Library rule and later decide that you need to modify the Library rule logic for use with a single object, you can convert the Library rule into an unnamed rule on the object. You can convert a Library rule into an unnamed rule whether the Library rule was applied in Design Manager or in Designer.

> **Caution:** You must convert the Library rule into an unnamed rule before you add any new logic. Converting a Library rule overwrites any existing rule logic for this rule. If you have already added logic to an unnamed rule and want to add Library rule logic to that existing logic, you must copy the logic and add it after the Library logic is imported.

To convert a Library rule into an unnamed rule:

1. Access the **Rule** tab for the object to which you want to apply the Library rule.

   For more information about accessing the **Rule** tab, see “Accessing the Rule Tab” on page 12.

2. If the advanced options do not appear on the **Rule** tab, click + to toggle the appearance of advanced options.

3. If a Library rule has not already been applied, do the following:

   - In the **Library rule** area, click ± .

     The **Select Rule** dialog box opens.

   - Select the Library rule you want to apply.

   - Click **OK**.

     The name and folder of the Library rule appear in the **Library rule** area.

4. Click −.

   The Library rule is converted into an unnamed rule.

2.4 Priority Rules

Priority rules are used with campaigns to set the order in which campaigns are included in customer output when space is limited. Setting a priority rule allows you to use customer data to determine this order. For example, a campaign that explains the advantages of bundling services from your company can be dynamically assigned a high or low priority based on whether the customer subscribes to more than one service, or based on information about specific services to which the customer subscribes. When the application is produced as output, campaigns are allowed to fill customer output, in order of priority, until all available design space is filled.

2.4.1 Creating a Priority Rule

This task assumes that you have previously created a campaign.

For information about creating a campaign, see *Managing Marketing Messages* in the Exstream Design and Production documentation.

To create a priority rule on a campaign:

1. In Design Manager, from the Library, drag the campaign to the Property Panel.
2. Click the **Priority** tab.
3. From the **Method for specifying priority of this campaign compared to other campaigns** drop-down list, select **Priority is set by a rule for each customer**.
4. Click in the **Priority rule** box.

   The **Rule** dialog box opens.
5. Create the rule logic.

   For more information about creating rule logic, see “Building the Logic for a Rule” below.
6. Click **OK**.

   The **Rule** dialog box closes.

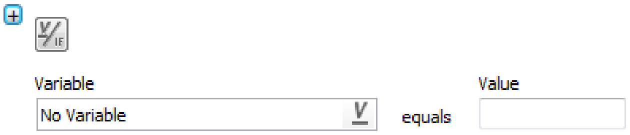
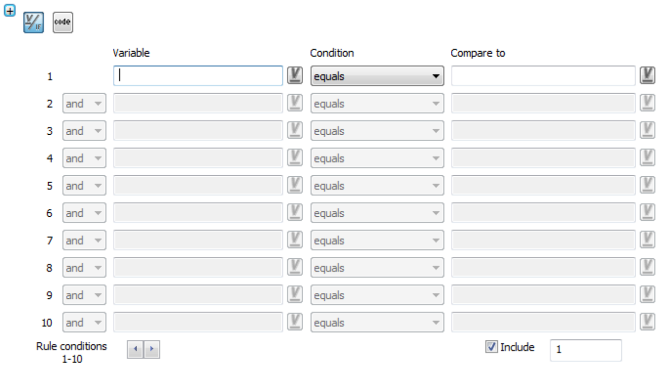
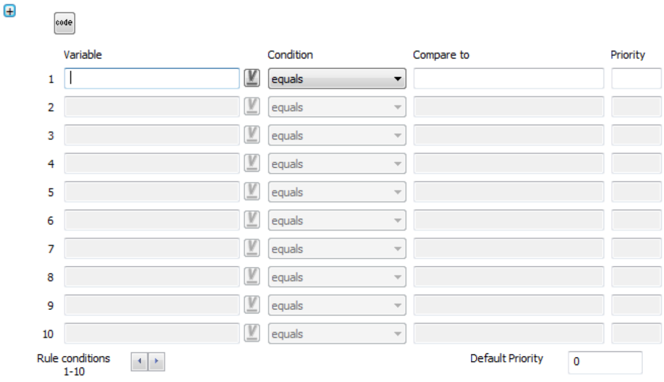
2.5 Building the Logic for a Rule

Exstream offers several different logic design views in which you can build the logic for a rule. Each view offers varying levels of design complexity, and you can select the view to use based on your needs. You are not restricted to any one method across Exstream. This versatile logic design ability lets you use any logic design view when creating logic for an object and lets you use different design views on different objects, if needed.

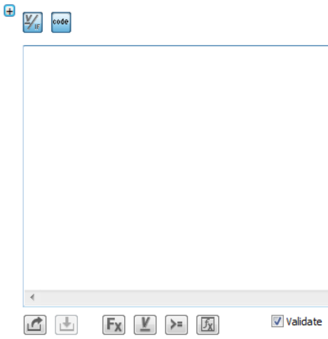
2.5.1 Logic Design Views in Which You Can Build Logic

You build rule logic on the **Rule** tab of an object. To begin creating logic, access the **Rule** tab for the object and select one of the following logic design views in which you want to build the logic for the rule:

Rule tab views

To do this	Use this view
Build basic rules where you are making a logic comparison between a single variable and a single static value For example, you want a document to be delivered only to customers in the United States.	**Simple equality rule view**  Variable No Variable equals Value For more information about creating logic in simple equality rule view, see "Building Rule Logic in Simple Equality Rule View" on page 22.
Build inclusion or exclusion rules where you are making a logic comparison between variables or groups of variables For example, you want a document to be delivered only to customers in the United States, but only if those customers are also in the state of Florida.	**Selection rule view**  Variable Condition Compare to 1 equals 2 and equals 3 and equals 4 and equals 5 and equals 6 and equals 7 and equals 8 and equals 9 and equals 10 and equals Rule conditions 1-10 ☑ Include 1 For more information about creating logic in selection rule view, see "Building Rule Logic in Selection Rule View" on page 23.
Build a rule for campaigns that determines which campaigns receive higher priority to be included in the final customer output, based on customer information For example, suppose you have a customer document that contains enough space to fit three campaigns, but the application currently contains five campaigns for customer distribution. You can set priorities to determine which campaigns are selected first and which campaigns are optional.	**Priority rule view**  Variable Condition Compare to Priority 1 equals 2 equals 3 equals 4 equals 5 equals 6 equals 7 equals 8 equals 9 equals 10 equals Rule conditions 1-10 Default Priority 0 For more information about creating logic in priority rule view, see "Building Rule Logic in Priority Rule View" on page 24.

Rule tab views, continued

To do this	Use this view
Build inclusion or exclusion rules where you require complex logic or calculations For example, you want a document to be delivered only to customers in the United States who are in the state of Florida and have an average household income less than $30,000.	**Code view**  ☑ Validate For more information about creating logic in code view, see “Building Rule Logic in Code View” on page 25.
Build, review, or edit logic in a third-party program	To build logic in a third-party program, you must start from the code view of the **Rule** tab and then export the rule to the third-party program. For more information about building logic in a third-party program, see “Building Rule Logic with a Third-Party Program” on page 27.

Toggling Between Rule Logic Design Views

When you access the rule properties for an object for the first time, the **Rule** tab automatically opens in simple equality rule view. When you open an existing rule, the **Rule** tab automatically opens in the last view used. For example, if you create rule logic for an object in code view, then when you return to the **Rule** tab of that object, the tab automatically opens in code view.

To toggle between rule logic design modes, do one of the following on the **Rule** tab:

Toggling between rule logic design modes

To	Do this
Toggle between selection rule view and simple equality rule view	Click V/IF .
Toggle between selection rule view or priority rule view and code view	Click code . **Tip:** By toggling to code view, you can see the code for the rule logic entered in any logic design view. Toggling to code view can be helpful if you want to review the code that is generated when you create rule logic in selection rule view or simple equality rule view.
Toggle between code view and simple equality rule view	Click V/IF .

Depending on the logic you create in each view, you might be restricted from returning to a simpler rule logic design view. Since each view offers more logic design capability, you might be restricted from using a simpler logic design view based on the logic you have created. For example, if you are in code view and your code includes multiple comparison statements (something that can't be done from simple equality rule view), then when you attempt to toggle to simple equality rule view, you receive an error message. In order to preserve your code, you are prevented from toggling to simple equality rule view.

2.5.2 Building Rule Logic in Simple Equality Rule View

By default, if there is no rule on an object, the **Rule** tab opens in simple equality rule view. This basic logic design view lets you select objects based on the value of a single variable. If the rule is proven true and the variable is equal to the value, then the object will be included in the customer output. For example, you might use simple equality rule view if you have a 'CustomerState' variable and you want only customers in the state of Texas to receive the object.

> **Tip:** If you want to exclude an object from customer output or you need a different type of variable/value comparison, consider using the selection rule view.

Keep in mind that you cannot use array variables in simple equality rules because these rules must equal a single value.

To create rule logic in simple equality rule view:

1. Access the **Rule** tab.

   For more information about accessing the **Rule** tab, see “Accessing the Rule Tab” on page 12.

2. If the **Rule** tab is not in simple equality rule view, click V/IF to toggle the view.

3. Click V and select a non-array variable.

4. In the **Value** box, enter the value the variable must match to be true.

   You must enter any string values in double quotation marks. For example, if the value of the variable is `TX`, you enter `"TX"`.

5. Click **OK**.

   The rule is applied to the object.

2.5.3 Building Rule Logic in Selection Rule View

The selection rule view allows you to set inclusion or exclusion rules by using logic comparisons between variables or groups of variables. This view allows you to create more complex or multi-faceted logic without having to create code.

To create a rule in selection rule view:

1. Access the **Rule** tab.

   For more information about accessing the **Rule** tab, see "Accessing the Rule Tab" on page 12.

2. If the **Rule** tab is not in selection rule view, click V to toggle the view.

3. Set one of the following inclusion methods to use if the rule logic is proven true:

To	Do this
Include the object when the rule is true	Select the **Include** check box.
Include an object multiple times when the rule is true (For rules on Design Manager objects only)	a. Select the **Include** check box. b. In the adjacent box, enter the number of times you want to include the object when the rule is true. **Note:** If you want to include messages multiple times and the adjacent box will not accept the value, your administrator might have restricted the number of times a message can be sent in a single document to a single customer. For information about changing administrator settings, see *System Administration* in the Exstream Design and Production documentation.
Use the group rule method to select specific rows or columns to include in customer output from a group of rows or columns	a. Select the **Include** check box. b. In the adjacent box, enter the number of times you want to include the object when the rule is true. For information on using the group rule method to select specific rows or columns in a table, see *Designing Customer Communications* in the Exstream Design and Production documentation.
Exclude the object from customer output when the rule is true	Clear the **Include** check box.

4. In the **Variable** box, click V and select a variable. This variable should contain the main values on which you want to base your rule. For example, if you want to include an object based on a customer's jurisdiction, the variable should contain jurisdiction information.

5. In the **Condition** box, select the condition you want to use when comparing variable

values. For **is like** comparisons, you can add additional code to make comparisons quicker or more precise.

For more information about the available "is like" conditions, see “"Is Like" Conditions” on page 191.

6. In the **Compare to**box, do one of the following:

   - Click V and select a variable. This should be the variable that contains the values you want to compare to the variable in the **Variable** box.
   - Enter a text string surrounded by double quotation marks.

   > **Caution:** With a string variable, anything not in quotation marks is considered the name of a variable. Always add double quotation marks on either side of a string, amount, date, or number.

7. To add additional sets of comparison information, repeat step 1 through step 5 as needed. You can enter up to 250 condition statements for a rule, but you can only view 10 at a time.

   You can click ◄ and ► to scroll through comparisons.

8. Select one of the following options to control the way in which condition statements interact:

If	Do this
Both of the adjoining condition statements must be true in order for the rule to be included/excluded from the customer output. For example, suppose you have entered two conditions to include a table: `CustomerState`equals `Kansas` and `CustomerCity` equals `Kansas City`. Only customers from Kansas City, Kansas, receive the table.	From the drop-down list adjacent to the second condition, select **and**.
Any of the adjoining condition statements can be true in order for the rule to be included/excluded from the customer output For example, suppose you have entered two conditions to include a table: `CustomerState`equals `Kansas` and `CustomerCity` equals `Kansas City`. The table is included in output if the customer is either from Kansas or resides in Kansas City, which could also include customers who reside in Kansas City, Missouri.	From the drop-down list adjacent to the second condition, select **or**.

9. Click **OK**.

   The rule is applied to the object.

2.5.4 Building Rule Logic in Priority Rule View

Priority rule view is used when you are creating a priority rule on a campaign. Using this logic design view, you can assign a priority level to a campaign based on customer information.

Inclusion settings are not used with priority rules because priority rules are by default set to include an object if the logic is proven true.

To create rule logic in priority rule view:

1. Access the **Rule** tab.

   For more information about accessing the **Rule** tab, see "Accessing the Rule Tab" on page 12.

2. If the **Rule** tab is not in priority rule view, click code to toggle the view.
3. In the **Default Priority** box, enter the number of the default priority for the campaign.
4. In the **Variable** box, click V and select a variable. This variable should contain the main values on which you want to base your rule. For example, if you want to include an object based on a customer's jurisdiction, the variable should contain jurisdiction information.
5. In the **Condition** box, select the condition you want to use when comparing variable values. For **is like** comparisons, you can add additional code to make comparisons quicker or more precise.

   For more information about the available "is like" conditions, see ""Is Like" Conditions" on page 191.

6. In the **Compare to** box, do one of the following:

   - Click V and select a variable. This should be the variable that contains the values you want to compare to the variable in the **Variable** box.
   - Enter a text string surrounded by double quotation marks.

   > **Caution:** With a string variable, anything not in quotation marks is considered the name of a variable. Always add double quotation marks on either side of a string, amount, date, or number.

7. Click **OK**.

   The rule is applied to the object.

2.5.5 Building Rule Logic in Code View

In code view, you can create rule logic using programming to create complex comparisons or add custom controls to the logic. Coding for logic in Exstream is similar to using Visual Basic, with some key differences.

For more information about the accepted programming practices in Exstream, see "Creating Effective and Efficient Code in Exstream" on page 189.

To create rule logic in code view:

1. Access the **Rule** tab.

   For more information about accessing the **Rule** tab, see "Accessing the Rule Tab" on page 12.

2. If the **Rule** tab is not in logic code view, click code to toggle the view.

3. In the Code Panel, enter code as needed. You can also use the V , >= , Fx, and fx buttons below the Code Panel to you select variables, conditions, and functions that are stored in your Exstream database and insert them directly into your code. Using the buttons to insert logic elements is especially useful for novice coders because using the buttons helps to make sure that code is entered accurately (for example, variables that are inserted using the buttons are always spelled correctly). To add logic elements directly into your code, complete any of the following as needed.

To	Do this
Insert a variable	a. Place the cursor in the Code Panel where you want to insert the variable b. Click V . The **Select Variable** dialog box opens. c. Select the variable you want. d. Click **OK**. The variable is inserted into the logic. For more information about variables, see *Using Data to Drive an Application* in the Exstream Design and Production documentation.
Insert a Library function	a. Place the cursor in the Code Panel where you want to insert the variable b. Click fx . The **Select Library Function or Subroutine** dialog box opens. c. From the **Type**drop-down list, select the type of Library function you want to add. d. From the **Folder**drop-down list, select the folder where you want to look for the Library function. e. In the **Functions** area, select the Library function you want. f. Click **OK**. The Library function is inserted into the logic. For more information about Library functions, see "Library Functions" on page 52.

To	Do this
Insert a built-in function	a. Place the cursor in the Code Panel where you want to insert the variable b. Click Fx . The **Select Built-in Function** dialog box opens. c. From the **Filter** drop-down list, select the type of built-in function you want to add. d. From the built-in function list, select the built-in function you want. e. Click **OK**. The built-in function is inserted into the logic. For more information about built-in functions, see "Built-in functions" on page 63.
Insert a condition	a. Place the cursor in the Code Panel where you want to insert the variable b. Click >= . The **Select a Condition** dialog box opens. c. Select the condition you want. d. Click **OK**. The condition is inserted into the logic.

4. Click **OK**.

   The rule is applied to the object.

2.5.6 Building Rule Logic with a Third-Party Program

If you have a program external to Exstream that you want to use to create or edit code, you can connect Exstream with the third-party program to allow you to more easily and accurately transfer code between Exstream and third-party program.

To use a third-party program:

1. If not previously set, specify the default third-party program you want to use for coding by doing the following:

   a. In Design Manager, from the Menu bar, select **Tools > Options**.

      The **Options** dialog box opens.

   b. In the **Data Mapping** area, in the **Text editor** box, click ▤ , and then select the EXE file for the program where you want to edit the code.

   c. Click **OK**.

      The **Options** dialog box closes.

   For information on setting the options for Design Manager and Designer, see *Getting Started* in the Exstream Design and Production documentation.

2. In Designer or Design Manager, access the **Rule** tab for the object to which you want to apply the code from a third-party program.

   For more information about accessing the **Rule** tab, see “Accessing the Rule Tab” on page 12.

3. Click ⤴.

   The rule opens in the third-party program. Any pre-existing code for this logic element is also exported and appears in the third-party program.

4. In the third-party program, make changes as needed to the code. Do not change the name of the file. In order for code to be imported properly when you are finished, you must use file name given by Exstream.

5. Save all changes.

6. In Designer or Design Manager, access the **Rule** tab for the rule you exported to the third-party program.

   For more information about accessing the **Rule** tab, see “Accessing the Rule Tab” on page 12.

7. Click ⤓.

   The code, along with your changes, is imported back into Exstream.

8. Click **OK**.

   The rule is applied to the object.

2.6 Setting Rule Timing for the Engine

During production, the engine has specific phases at which specific actions are performed on objects in an application. This process is commonly referred to as engine timing. The engine reads information about building your application in a specific order during an engine run. Since rules can be applied to objects that are read or created at specific times during an engine run, it is important to time when your rules are read by the engine so that the correct information will be available at the time when the rule is read. For example, suppose your rule is applied to a table that is set to late compose, but your rule contains variables that are also set to late compose. Because the table is composed on the page before the engine has the information needed to exclude the object, you might receive unexpected results in the customer output.

2.6.1 Delaying Rules on Documents to be Executed at the End of Customer Processing

By default, rules are executed at the time when the object to which the rule is applied is composed by the engine. With documents, you can delay rules for the inclusion or exclusion of the document until after all other processing is complete. For example, you might want to determine whether a customer receives a document based on the customer's account totals, which will not be known until the end of processing.

You should delay the rule run time of a document only when necessary. By default, all documents are composed at customer run time and removed at the queue level after they are composed. When you delay the rule run time of the document, the document is composed at customer run time; however, it is not removed from the output queue until after all other processing is complete and after the "At end" rule is executed. Applications that contain many documents that use "At end" rule run time can increase processing time and decrease the efficiency of your engine run.

For information on engine processes, see *Preparing Applications for Production* in the Exstream Design and Production documentation.

To delay rules on documents to be executed at the end of customer processing:

1. In Design Manager, from the Library, drag the document to the Property Panel.
2. Click the **Targeting** tab.
3. From the **Rule run time** drop-down list, select **At end**.
4. From the Standard toolbar, click 💾.

   The document is saved.

2.6.2 Setting the Timing on Live Objects

Because end users of Live applications might be able to interact with many of the objects to which you have applied rules, and because end users might change the data used by your rule, it is important to set additional timing constraints on rules that apply to your Live application.

To specify the time, or times, when the rule runs for Live applications:

1. Access the **Rule** tab for the Live object for which you want to set the timing.

   For more information about accessing the **Rule** tab, see “Accessing the Rule Tab” on page 12.
2. If the advanced options do not appear on the **Rule** tab, click + to toggle the appearance of advanced options.

3. From the **Execute time** drop-down list, select the time at which you want the rule to be read by the engine.

To execute rules at this time	Do this
Execute rules according to the timing specified in the **Default variable substitution & rule execution time** drop-down list on the edit settings	Select **Use Edit Settings default**.
Execute rules only during the first engine run when the interactive document is produced This means that users cannot upload and use their own distribution lists.	Select **Initial Engine run only**.
Execute rules during the first engine run and when users upload distribution lists in the LiveEditor (previewing their data)	Select **Initial Engine and Interactive Editor**.
Execute rules during the first engine run, the last engine run (usually to produce output in a format other than DLF), and when users upload mailing lists in the LiveEditor (previewing their data)	Select **Initial Engine, Interactive Editor, and Final Engine**.

For more information on edit settings for Live objects, see *Designing for LiveEditor* in the Exstream Design and Production documentation.

4. Click **OK**.

The Live timing is applied to the selected object.

2.7 Testing or Troubleshooting Rules

If you find that you need to troubleshoot rules, Exstream offers the following tools you can use to help track down any issues:

- **Validation**—Use validation to verify that code you enter for rules is valid.
- **Code trace**—Use code trace to record each line of source code as it is executed by the engine and, depending on your settings, return additional information about the logic.

2.7.1 Validating the Code Used in the Rule Logic

As you create code, you can turn on validation to make sure that you have entered code correctly before exiting or saving the rule.

To turn on validation for rule logic:

1. Access the **Rule** tab.

   For more information about accessing the **Rule** tab, see “Accessing the Rule Tab” on page 12.

2. Select the **Validate** check box.

3. When you click **OK** to close or save the rule, a validation check is run on the rule logic.

4. If there are errors in the code, an **Errors** dialog box opens, showing a list of errors or warnings for code entered in the Code Panel.

5. To locate a specific error in the code, select an error from the list and click **OK**.

   The **Errors** dialog box closes and the cursor in the code box moves to the location of the selected error. If no error was selected, the cursor moves to the location of the first error in the list.

   > **Tip:** To see additional information about an error, you can locate the error by error number in the message dictionary.

   For information on the message dictionary, see *Preparing Applications for Production* in the Exstream Design and Production documentation.

6. Fix any errors issued by the **Errors** dialog box.

7. Click **OK**.

   Validation is run on the code again. If no errors are encountered, the rule is saved and you return to the object properties. If additional errors are found, repeat step 5 through step 7 as needed.

2.7.2 Setting a Code Trace Filter on a Rule

You can trace code to collect information about what is happening to the logic as the engine processes your application. This information can be helpful if you need to test or troubleshoot the timing at which the code is read by the engine, or if you are receiving unexpected results. Before you can run code trace on logic, you must set a filter that specifies the amount of information you want the code trace to return on the logic you are testing.

To set a code trace filter on a rule:

1. Access the **Rule** tab.

   For more information about accessing the **Rule** tab, see “Accessing the Rule Tab” on page 12.

2. If the advanced options do not appear on the **Rule** tab, click + to toggle the appearance of

advanced options.

3. From the **Code trace** drop-down list, select the code trace filter for the level of information you want:

To	Do this
Write each line of the formula to the debug file as it is executed	Select **Source Line**.
Write each line of the formula and the names of variables to the debug file as they are used	Select **Assignment**.
Write each line of the formula and the value of variables to the debug file as they are used	Select **All Variables**.

4. From the Standard toolbar, click 💾.
5. Repeat step 1 through step 4 on any additional rules you want to trace.
6. Run Trace/Watch/Debug to view a report on code where you applied a code trace.

   For more information on running Trace/Watch/Debug, see *Preparing Applications for Production* in the Exstream Design and Production documentation.

2.8 Reviewing the Rules Applied to Your Design

If you want to review all the business logic that has been applied to a design, you can run a design rule report. These reports can be generated for any design you work with in Designer, such as pages, messages, section objects, and paragraph objects. Design rules reports are useful for helping you avoid potential logic conflicts in your design.

The returned report includes information about rules applied to all objects in the current design layer of your design, with the exception of rules applied to variables. Design rules reports let you review and compare all the rules applied to your design at one time, instead of reviewing rules individually by selecting each object.

To generate a design rules report:

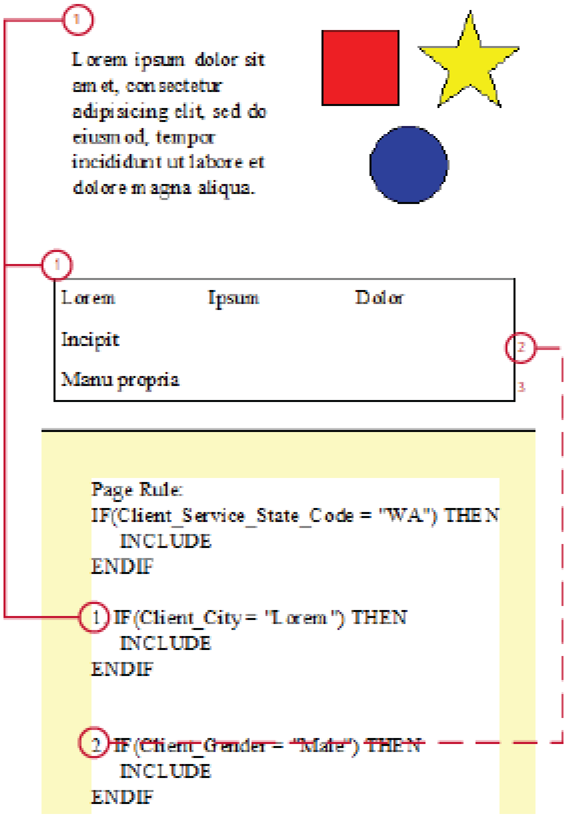
1. Open the object you want to review in Designer.
2. If you use design layers or language layers, make sure you are viewing the correct layer.
3. From the Menu bar, select **Tools > Design Rules Report**.

   The report is automatically generated and appears below your design.

> **Tip:** Because the design rules report uses the default font for your database, the report might not appear as expected when you enable complex text layout in your database (in the **System Settings**, on the **Text and Fonts** tab, the **Enable complex text layout** check box).
>
> If you are generating a design rules report for a language layer that uses complex text layout, characters in the report can sometimes appear incorrectly. In order for the design rules report to appear as expected in this case, you must temporarily change the default font for your database to the font used in the rules on the language layer for which you want to run the design rules report. (You set the default font for your database in the **System Settings**, on the **Text and Fonts** tab, in the **Default font** section.) After you generate the report, be sure to change the default font back to its previous setting.
>
> For information about using complex text layout, see *System Administration* in the Exstream Design and Production documentation.

After the report is generated, an index number appears next to each object on which a rule is applied. This index number corresponds to a rule in the design rules report below the design. Unnamed rules are numbered and included in the design rules report individually, but if you use a Library rule on multiple objects in a single design, all the objects that reference the same Library rule have the same index number. For design rules reports run on section objects, the index entry also includes the names of the section objects and paragraph objects which share the rule, since a section object can contain multiple section objects and paragraph objects.

Example of design rules report on a page



Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod, tempor incididunt ut labore et dolore magna aliqua.

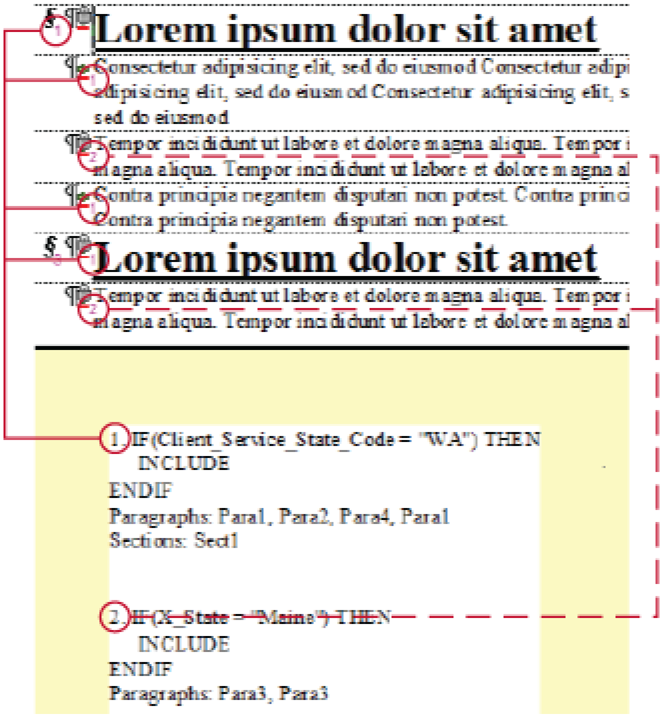
Lorem	Ipsum	Dolor
Incipit		
Manu propria		

```
Page Rule:
IF(Client_Service_State_Code = "WA") THEN
    INCLUDE
ENDIF

1 IF(Client_City = 'Lorem") THEN
    INCLUDE
ENDIF


2 IF (Client_Gender = "Male") THEN
    INCLUDE
ENDIF
```

Example of design rules report on a section



Lorem ipsum dolor sit amet

Consectetur adipisicing elit, sed do eiusmod Consectetur adipi
adipisicing elit, sed do eiusmod Consectetur adipisicing elit, s
sed do eiusmod

Tempor incididunt ut labore et dolore magna aliqua. Tempor i
magna aliqua. Tempor incididunt ut labore et dolore magna al

Contra principia negantem disputari non potest. Contra princi
Contra principia negantem disputari non potest.

Lorem ipsum dolor sit amet

Tempor incididunt ut labore et dolore magna aliqua. Tempor i
magna aliqua. Tempor incididunt ut labore et dolore magna al

```
1. IF(Client_Service_State_Code = "WA") THEN
     INCLUDE
   ENDIF
   Paragraphs: Para1, Para2, Para4, Para1
   Sections: Sect1

2. IF (X_State = "Maine") THEN
     INCLUDE
   ENDIF
   Paragraphs: Para3, Para3
```

Design rules reports are automatically reset when you close Designer, to prevent design rules reports from being included in production output. However, you can print the report locally by selecting **File > Print** from the Menu bar. The printed version includes all the text and formatting.

Chapter 3: Formulas

Formulas allow you to take customer information and use it to calculate new values without modifying the original information. For example, you can create a formula that calculates the total amount a customer owes by adding all the charges a customer has made for the month.

Formulas, are not just used with numerical calculations, however. Formulas can also be used to manipulate customer data so that it can be used differently. For example, suppose your data contains customer state and customer city information which is combined into a single entry, such as `KYFrankfort.` You can build a formula to divide this information into local variables so that the information can be used separately in the customer output as `KY` and `Frankfort`, while preserving the original `KYFrankfort` entry in your original data.

All formula logic is stored in a formula variable. This allows formulas to be placed anywhere in an application where you can use variables.

To create a formula, you must complete the following tasks:

1. "Creating a Formula Variable" below
2. "Building the Logic for a Formula" on the next page
3. "Setting Formula Timing for the Engine" on page 41
4. "Applying Formulas to an Application" on page 48

You can also complete the following optional task as needed:

- "Testing or Troubleshooting Formulas" on page 49

3.1 Creating a Formula Variable

A formula variable can be referenced in the same way you reference any variable. You build all the logic in a formula variable object that can be inserted into the text, into other logic, or into any other location where you can use variables in Exstream.

To create a formula variable:

1. In Design Manager, in the Library, right-click the **Data Dictionary** heading, and, from the shortcut menu, select **New Variable**.

   The **New Variable** dialog box opens.

2. In the **Name** box, enter a name. In the **Description** box, enter a description (optional).
3. From the **Type** drop-down list, select the type of information that will be returned by the formula variable.

> **Note:** If your formula will result in a value over 2 billion, do not select **Integer** from the **Type** drop-down list.

For more information about variable types, see *Using Data to Drive an Application* in the Exstream Design and Production documentation.

4. If the variable is an array, select the **Array** check box.
5. In the **Design sample** box, enter a sample of what the values in this variable might be. For example, if you are creating an integer type variable, your design sample might be `123`. This design sample can be seen as a placeholder for the variable when you toggle the sample view of the variables in Designer by clicking V/A.
6. Click **Finish**.

   The variable opens in the Property Panel for you to define.
7. Click the **Basic** tab.
8. From the **Source** drop-down list, select **Formula**.
9. Click the **Values** tab.
10. In the **Formula** area, enter the logic for the formula. For example you might enter the computation of a value or a counter.

    For more information about building the logic for a formula, see “Building the Logic for a Formula” below.
11. From the Standard toolbar, click 💾.

    The formula is saved.

3.2 Building the Logic for a Formula

Formula logic uses generally accepted programming practices modeled after Visual Basic.

For more information about the accepted programming practices in Exstream, see “Creating Effective and Efficient Code in Exstream” on page 189.

As a best practice, you can group formulas by compute time to improve processing. For example, if an application contains multiple formula variables that are computed for each customer, you can place the code for all formulas into a single formula variable, and set user value variables from there. Grouping formulas also makes it easier to collectively change the engine timing of formulas. For example, it is easier to change the engine timing of one variable that contains 40 computations than it is to change the engine timing of 40 separate variables. You cannot combine formulas that use customer data and section data. Since these two types of data are timed differently in the engine, combining them in a single variable could cause unexpected results.

For information on compute times, see *Preparing Applications for Production* in the Exstream Design and Production documentation.

To create formula logic:

1. In Design Manager, from the Library, drag the formula variable to the Property Panel.
2. Click the **Values** tab.
3. In the **Formula** area, enter code as needed. You can also right-click the **Formula** area to select variables, conditions, and functions that are stored in your Exstream database and insert them directly into your code. Inserting logic elements by using the shortcut menu is especially useful for novice coders, because using the shortcuts helps to make sure that code is entered accurately. For example, variables that are inserted using the shortcut are always spelled correctly. To add logic elements directly into your code, complete any of the following as needed.

To	Do this
Insert a variable	a. Place the cursor in the **Formula** box where you want to insert the variable. b. Right-click, and, from the shortcut menu, select **Variable**. The **Select Variable** dialog box opens. c. Select the variable you want. d. Click **OK**. The variable is inserted into the logic. For more information about variables, see *Using Data to Drive an Application* in the Exstream Design and Production documentation.
Insert a Library function	a. Place the cursor in the **Formula** box where you want to insert the Library function. b. Right-click, and, from the shortcut menu, select **Library Function**. The **Select Library Function or Subroutine** dialog box opens. c. From the **Type** drop-down list, select the type of Library function you want to add. d. From the **Folder** drop-down list, select the folder where you want to look for the Library function. e. In the **Functions** area, select the Library function you want. f. Click **OK**. The Library function is inserted into the logic. For more information about Library functions, see "Library Functions" on page 52.

To	Do this
Insert a built-in function	a. Place the cursor in the **Formula** box where you want to insert the built-in function. b. Right-click, and, from the shortcut menu, select **Built-in Function**. The **Select Built-in Function** dialog box opens. c. From the **Filter** drop-down list, select the type of built-in function you want to add. d. From the built-in function list, select the built-in function you want. e. Click **OK**. The built-in function is inserted into the logic. For more information about built-in functions, see "Built-in functions" on page 63.
Insert a condition	a. Place the cursor in the **Formula** box where you want to insert the condition. b. Right-click, and, from the shortcut menu, select **Syntax**. The **Select a Condition** dialog box opens. c. Select the condition you want. d. Click **OK**. The condition is inserted into the logic.

4. If your formula will return a value, you must have one of the following prefixes at the beginning of the code for the formula. If you combine multiple formulas into a single formula variable, each separate calculation must have its own prefix.

If you use this type of variable	Use this prefix

Static variable (non-array)	Use one of the following prefixes:
	• `Value =`
	• `Integer =`
	• `[variable name] =`
	If you are using the variable as a counter variable, do not use the `Value=`prefix or you can receive incorrect results.
	For example:
	`[variable name]=[variable name] + 1`
	A completed example might read as follows:
	`Total=Total + 1`

If you use this type of variable	Use this prefix

Array variable	Use one of the following prefixes:
	• `Value(i)=`
	• `Integer(i)=`
	• `[variable name](i)=`
	With these prefixes, `i` is equal to the number of elements you want to use.
	You can use any of the above prefixes if you are using the variable as a counter variable. The subscript `i` acts as a counter.
	For example:
	`Integer i`
	`Value(i) = <CreditCardAmount> + <DebitCardAmount>`
	`NEXT i`

> **Note:** If you receive the error message `#5135, Function returns a value`, make sure you included one of the prefixes in front of the formula.

5. From the Standard toolbar, click 💾.

   The formula is saved.

   For tips on creating efficient code, see “Creating Effective and Efficient Code in Exstream” on page 189.

3.2.1 Building Formula Logic with a Third-Party Program

If you have a program external to Exstream that you want to use to create or edit code, you can connect Exstream with the third-party program to allow you to more easily and accurately transfer code between Exstream and third-party program.

To use a third-party program you must do the following:

1. If not previously set, specify the default third-party program you want to use for coding:

   a. In Design Manager, from the Menu bar, select **Tools > Options**.

      The **Options** dialog box opens.

   b. In the **Data Mapping** area, in the **Text editor** box, click 🗎 and then select the EXE file for the program where you want to edit logic code.

c. Click **OK**.

   The **Options** dialog box closes.

   For information about setting personal options in Exstream, see *Getting Started* in the Exstream Design and Production documentation.

2. In Design Manager, from the Library, drag the formula variable to the Property Panel.
3. Click the **Values** tab.
4. Click 📤.

   The logic element opens in the third-party program. Any existing code for this logic element is also exported and appears in the third-party program.

5. In the third-party program, make changes as needed to the code. Do not change the name of the file. In order for code to be imported properly when you are finished, you must use file name given by Exstream.
6. Save all changes.
7. In Design Manager, from the Library, drag the formula variable to the Property Panel.
8. Click the **Values** tab.
9. Click 📥.

   The code, along with your changes, is imported back into Exstream.

10. From the Standard toolbar, click 💾.

    The formula is saved.

3.3 Setting Formula Timing for the Engine

During production, the engine has specific phases at which specific actions are performed on objects in an application. This process is commonly referred to as engine timing. The engine reads information about building your application in a specific order during an engine run. Formulas require you to set two separate timings to help you to receive the values you need at the time you need them in the final engine run:

- **Reset time**—The time at which the values that are used to make calculations in the formula are reset, to allow new or updated customer data to be applied to the logic
- **Compute time**—The time at which the value is written into the customer output

3.3.1 Setting a Reset time on a Formula Variable

Different customer information is available at different times during the engine run. Objects in the application can also be set to be processed at specific times during the engine run. Because you have this ability to set engine timing on objects, it is important that you set variable values to reset at the correct time to give you the results you want. Resetting a variable is similar to pressing the clear button on a calculator to start a new calculation. After a variable is reset, new information is pulled from the data to complete the calculation.

The reset time allows you to set how frequently you want new values to be used in logic of the formula. Also take in consideration how frequently you want the variable to compute.

For example, suppose you are calculating a grand total of all spending across a customer's accounts. You would want the variable values to be reset for each customer so that only the values from a single customer are included in the calculation. However, suppose you are calculating a sub-total of each account held by a customer and each account is stored using data sections in the customer driver file. In this case, you would want the variable values to be reset before each data section in the customer data.

To set a reset time on a formula variable:

1. In Design Manager, from the Library, drag the formula variable to the Property Panel.
2. Click the **Basic** tab.
3. From the **Reset time** drop-down list, specify when the variable's value is reset and a new value is calculated. If your variable is an array, the reset time controls when the variable is reset to zero elements. Select from the following options:

To	Do this
Reset the variable before each customer	Select **Automatically**. If the variable is mapped in a section-based data file, the variable also resets before any section is read. This is the default.
Reset the variable only before each customer is read, regardless of section-based data	Select **Before each customer**.
Reset the variable only at the beginning of each engine run. This causes the variable to have a single value across all customers.	Select **Never reset**. **Caution:** Do not use **Never reset** for formulas that represent a unique value for each customer.
Reset the variable before a named section in the data file. This lets you create subtotals before a named section in the data file.	a. Select **Named section**. b. In the adjacent box, enter the section name exactly as it is specified in the data file.

To	Do this
Reset the variable before all sections, regardless of name	Select **All sections**.

4. From the Standard toolbar, click 💾.

   The formula is saved.

3.3.2 Setting a Compute Time on a Formula Variable

While designing objects for your application, you also have control over when some objects are composed by the engine. Because you have this ability to set engine timing on objects, it is important that you set variable values to be computed at the correct time to give you the results you want. Setting a compute time controls how frequently you want to receive a value from the formula. This is similar to pressing equal on a calculator to receive a value.

When setting a compute time, also take into consideration how frequently you want the formula logic values to be reset, since these values can affect the final value you receive when the variable is computed.

For more information about controlling how frequently the values within a formula are reset, see “Setting a Reset time on a Formula Variable” on the previous page.

To set a compute time on a formula variable:

1. In Design Manager, from the Library, drag the formula variable to the Property Panel.
2. Click the **Values** tab.
3. From the **Compute time** drop-down list, specify when you want the formula to compute. Select from the following options:

To calculate at this time	Do this
When the variable is encountered as referenced by an object	Select **As-needed**. **Caution:** If the variable will be placed on late compose object (an object that has the **When to compose** setting of **Late Rule**), do not use **As-needed** as a compute time for the variables. Formulas that have a **Compute time** of **As-needed** cannot contain references to themselves. For example: `x=x+1` where `x` is the variable name. **Tip:** For variables in schema model data files, it is a best practice to select a **Compute time** of **As-needed**. This option optimizes the way in which the engine accesses variable data in schema model data files, because for schema model data files, all of a customer's data is made available at the beginning of the processing of that customer. This data availability means that variables in schema model data files are best computed as they are encountered in the design. For more information about schema model data files, see *Using Data to Drive an Application* in the Exstream Design and Production documentation.
For each data section	Select **All Sections**.
After processing the last customer	Select **Completion**.
For each customer, before any section data is read	Select **Customer**.
Before processing the first customer	Select **Initialization**.
After the engine reads a specific data section	a. Select **Named Section**. b. In the box below the drop-down list, enter the data section name exactly as it appears in the data file.
After the data aggregation event	Select **Data aggregation**.
Only in a postsort processing run	Select **Don't compute**.
For each page	Select **Page**.
For each customer, after all section data has been read	Select **Post-Customer**.

4. If the application uses output sorting and bundling, select the timing at which the formula computes during a postsort processing run from the **Post-sort compute time** drop-down list. Select from the following options:

To calculate at this time	Do this
When the formula is encountered as referenced by an object	Select **As-needed**. **Tip:** For variables in schema model data files, it is a best practice to select a **Post-sort compute time** of **As-needed**. This option optimizes the way in which the engine accesses variable data in schema model data files, because for schema model data files, all of a customer's data is made available at the beginning of the processing of that customer. This data availability means that variables in schema model data files are best computed as they are encountered in the design. For more information about schema model data files, see *Using Data to Drive an Application* in the Exstream Design and Production documentation.
For each page	Select **Page**.
For each customer, after all section data has been read	Select **Post-Customer**.
Before processing the first customer	Select **Initialization**.
After processing the last customer	Select **Completion**.
Only in a pre-processing run	Select **Don't compute**.

For information on the Output Sorting and Bundling module, see *Creating Output* in the Exstream Design and Production documentation.

> **Note:** The following is a compute time consideration if you are upgrading an Exstream database from version 4.0 or earlier to version 5.0 or later: if you have a **Compute time** setting of **Post-Customer** or **Completion** specified as **Compute time**, the **Post-sort compute time** setting is automatically changed to **Don't compute** when you upgrade.

5. From the Standard toolbar, click 💾.

   The formula is saved.

3.4 Eliminating Repetition in Section Data Using Formula Logic and Data Aggregation

You can use data aggregation, a process of grouping and manipulating data sections, to eliminate unneeded or repeated data sections. This process allows you to streamline the use of section-driven documents, objects, and content in the customer output. The data aggregation process is completed during the data aggregation engine timing, before individual customer data is read, to allow the customer output to reflect any changes to section-driven content that were caused by data aggregation.

Using a formula, you can control which data sections are affected by data aggregation changes. For example, in the section data, you might have some repeated data sections that you want to keep, while other data sections can be combined or deleted. Within the formula, you can use data section built-in functions to control how the engine reads, manipulates, and deletes duplicate section data.

For example, suppose you have the following data structure for an insurance policy:

```
"Deductible"
   Plan: A
   Value: 500
"Deductible"
   Plan: B
   Value: 500
"Deductible"
   Plan: C
   Value: 1000
```

By using data aggregation settings to force the areas in the data structure to combine, you can create the following data structure:

```
"Deductible"
   Plan: A, B
   Value: 500
"Deductible"
   Plan: C
   Value: 1000
```

To set up data aggregation with formula logic, you must complete the following tasks:

1. “Mapping Section Data for Data Aggregation in a Customer Driver File” on the next page
2. “Controlling How Data is Manipulated by Data Aggregation Using a Formula” on the next page

You can also use the formula-driven method of data aggregation in combination with data aggregation settings on paragraph and section objects. However, if you use both methods of data aggregation, any aggregation controls set by the formula are computed only after the paragraph and section aggregation controls.

For more information about setting up data aggregation with paragraph and section objects, see *Designing Customer Communications* in the Exstream Design and Production documentation.

For more information about data sections, see *Using Data to Drive an Application* in the Exstream Design and Production documentation.

3.4.1 Mapping Section Data for Data Aggregation in a Customer Driver File

You cannot use flat files to drive data aggregation. Data must be mapped in a customer driver file, in XML format, which requires you to license the XML/JSON Input module. Avoid using empty tags in the data file.

For sections in the data to be merged, the data in each section must be balanced. For example, the following data structure allows data sections to merge:

```
"Deductible"
   Plan: A
   Value: 500
"Deductible"
   Plan: B
   Value: 500
```

In contrast, data structured in the following manner does not allow the data sections to merge because it is not balanced:

```
"Deductible"
   Plan: A
   Value: 500
      "Out of Pocket Max"
         Value: 2500
"Deductible"
   Plan: B
   Value: 500
```

For more information about setting up a customer driver file in the XML format, see *Using Data to Drive an Application* in the Exstream Design and Production documentation.

3.4.2 Controlling How Data is Manipulated by Data Aggregation Using a Formula

Using formula logic to drive data aggregation requires a combination of formula logic and data section built-in functions. The built-in functions allow you to control the way data is moved or combined. Since data aggregation must occur during the data aggregation engine timing, the formula must be set to allow the built-in functions to be computed at the correct time during engine processing.

For more information about how data aggregation affects engine timing, see *Preparing Applications for Production* in the Exstream Design and Production documentation.

1. Create a formula variable.

   For more information about data sections, see *Using Data to Drive an Application* in the Exstream Design and Production documentation.

2. Click the **Values** tab.

3. In the **Formula** area, enter the logic you need to control how data is manipulated during data aggregation. You can use any of the data section functions and any additional logic, as needed.

   For more information about how data aggregation affects engine timing, see “Data section functions” on page 85.

4. From the **Compute time** drop-down menu, select **Data aggregation**.

5. From the Standard toolbar, click 💾.

   The formula is saved.

3.5 Applying Formulas to an Application

Formulas can be applied to an application in one of the following ways:

- Insert a formula variable directly into the text in your design.
- Apply a formula variable anywhere you can use variables.
- Insert the formula variable into the code of another logic element.
- Reference the formula from the application. You must use this method only if the formula is not applied to the application using one of the previous methods.

For more information about applying variables to your design, see *Using Data to Drive an Application* in the Exstream Design and Production documentation.

3.5.1 Referencing a Formula Variable from an Application

If the formula variable you create is not used directly on a page, you must reference it in the application. If the formula is used as a subroutine in another logic object, you must place the formula in the application in the Library.

To reference a formula variable from an application:

1. In Design Manager, from the Library, drag the application to the Property Panel.
2. Click the **Dynamic Objects** tab.
3. In the **Application variables** area, click ➕.

   The **Select Variables** dialog box opens.
4. Select the formula variable and click **OK**.

   The variable is added to the **Application variables** area.
5. From the Standard toolbar, click 💾.

   The formula is saved.

For more information about adding variables to applications and pages, see *Using Data to Drive an Application* in the Exstream Design and Production documentation.

3.6 Testing or Troubleshooting Formulas

If you find that you need to troubleshoot formulas, Exstream offers the following tools you can use to help track down any issues:

- **Validation**—Use validation to verify that a formula returns a valid value.
- **Code trace**—Use code trace to record each line of source code as it is executed by the engine and, depending on your settings, return additional information about the logic.

3.6.1 Validating the Value that is Returned by a Formula

As you create code, you can turn on validation to make sure that you have entered code correctly. You can perform this validation before exiting or saving the formula variable.

To turn on validation for formula logic:

1. In Design Manager, from the Library, drag the formula variable to the Property Panel.
2. Click the **Basic** tab.
3. Select the method you want to use to validate the value of the formula variable. Select from the following options:

If the return value should be this	Do this
Within a specific range	a. From the **Validation method** drop-down list, select **Value range**. b. In the boxes below, enter the beginning number and ending number for the value range.
Equal to a specific value	a. Select **Equal to**. b. In the box below, enter the specific value the formula variable value should equal.
Greater than a specific value	a. From the **Validation method** drop-down list, select **Greater than**. b. In the **Valid values** boxes, enter the specific value the formula variable value should be greater than. If you select this option, the value cannot be equal to the number you enter.
Greater than or equal to a specific value	a. From the **Validation** method drop-down list, select **Greater than or equal to**. b. In the **Valid values** box, enter the specific value the formula variable value should be greater than or equal to.
Less than a specific value	a. From the **Validation method** drop-down list, select **Less than**. b. In the **Valid values** box, enter the specific value the formula variable value should be less than. If you select this option, the value cannot be equal to the number you enter.
Less than or equal to a specific value	a. From the **Validation method** drop-down list, select **Less than or equal to**. b. In the **Valid values** box, enter the specific value the formula variable value should be less than or equal to.
Anything except for a specific value	a. From the **Validation method** drop-down list, select **Not equal to**. b. In the **Valid values** box, enter the specific value the formula variable value should not be equal to.
Match the value of a Library function	a. From the **Validation method** drop-down list, select **Function**. b. In the **Validation function** box, select the Library function that should have a value that matches the value of the formula variable.

3.6.2 Setting a Code Trace Filter on a Formula

You can trace code to collect information about what is happening to the logic as the engine processes your application. This information can be helpful if you need to test or troubleshoot the timing at which the code is read by the engine, or if you are receiving unexpected results. Before you can run code trace on logic, you must set a filter that specifies the amount of information you want the code trace to return on the logic you are testing.

To set a code trace filter on a formula:

1. In Design Manager, from the Library, drag the formula to the Property Panel.
2. Click the **Values** tab.
3. From the **Code trace** drop-down list, select the code trace filter you want to use for the formula:

To	Do this
Write each line of the formula to the debug file as it is executed	Select **Source Line**.
Write each line of the formula and the names of variables to the debug file as they are used	Select **Assignment**.
Write each line of the formula and the value of variables to the debug file as they are used	Select **All Variables**.

4. From the Standard toolbar, click 💾.
5. Run a code trace report.

   The report contains the information you requested for the logic.

For information about running a code trace report, see *Preparing Applications for Production* in the Exstream Design and Production documentation.

Chapter 4: Library Functions

Library functions are special pieces of code you can create which cause an action to occur based on information in a customer's data. For example, you can use a Library function to open a specific reference file if a customer has a certain type of account. Library functions are also used to analyze data and return the findings. For example, you can use a Library function to return only a customer's anniversary month, even if the month, day, and year are listed in the data.

Exstream also offers built-in functions, which are previously created functions that perform common or general functions. However, if you need more customized functionality, you can also create your own functions in Design Manager using Visual Basic-style code.

To create a Library function, you must complete the following tasks:

1. "Creating a Library Function Object" below
2. "Building the Logic for a Library Function" on page 54
3. "Using Library Function Arguments" on page 57 (as needed)
4. "Applying Library Functions and Subroutines to an Application" on page 61

You can also complete the following optional task as needed:

- "Testing or Troubleshooting Library Functions" on page 61

4.1 Creating a Library Function Object

1. In Design Manager, in the Library, right-click the **Functions** heading, and, from the shortcut menu, select **New Function**.
2. In the **Name** box, enter a name. As with variables, you cannot make a Library function with a duplicate name, nor can the name have spaces or non-alpha or numeric characters in it. If the name does not meet these criteria, you receive an error message. If you rename a function, all references to that function are updated to reflect the change in the name.

> **Note:** When working with Library functions, keep in mind that the SBCS engine cannot process functions whose names contain DBCS characters. Functions that are used in both SBCS and DBCS applications must have names that are made up of SBCS characters only.
>
> Additionally, the SBCS engine cannot correctly process functions that reference the following:
>
> - Variables whose names contain DBCS characters
> - Variables that contain DBCS content
> - DBCS string literals

3. In the **Description** box, enter a description (optional).

   > **Caution:** Do not create a Library function with the same name as a variable. If you do, the variable is used instead of the function when the name is referenced.

4. Click **Finish**.

   The new function opens in the Property Panel for you to define.

5. In the **Type** area, specify the type of function you are creating. Select one of the following options:

   - **Function**—Analyzes data and returns a value
   - **Subroutine**—Does not return a value. Instead, it performs an action based on data.

6. If you select the **Function** radio button, the **Data Type** drop-down list lets you specify the type of data the function returns. Select from the following options:

   - **String**—Text
   - **Integer**—Whole number
   - **Boolean**—True or false
   - **Floating**—Numeric value with decimals
   - **Date**—Date (with or without time of day)
   - **Currency**—Monetary amount

   > **Caution:** Exstream does not support exceptions for overflows or underflows. If your math results in values of more than 2 billion, do not select a **Data Type** option **Integer**.

7. From the Standard toolbar, click 💾.

   The Library function is saved.

4.2 Building the Logic for a Library Function

Library functions use generally accepted programming practices modeled after Visual Basic.

For more information about the accepted programming practices in Exstream, see “Creating Effective and Efficient Code in Exstream” on page 189.

To create Library function logic:

1. In Design Manager, from the Library, drag the Library function to the Property Panel.

2. In the **Logic** area, enter code as needed. You can also use the V, >=, Fx, and fx buttons below the **Logic** box to select variables, conditions, and functions that are stored in your Exstream database and insert them directly into your code. Using the buttons to insert logic elements is especially useful for novice coders because using the buttons helps to make sure that code is entered accurately (for example, variables that are inserted using the buttons are always spelled correctly). To add logic elements directly into your code, complete any of the following as needed

To	Do this
Add a variable	a. Click V . The **Select Variable** dialog box opens. b. Select the variable you want. c. Click **OK**. The variable is inserted into the logic. For more information about variables, see *Using Data to Drive an Application* in the Exstream Design and Production documentation.

To	Do this
Add a Library function	a. Click fx . The **Select Library Function or Subroutine** dialog box opens. b. From the **Type** drop-down list, select the type of Library function you want to add. c. From the **Folder** drop-down list, select the folder where you want to look for the Library function. d. In the **Functions** area, select the Library function you want. e. Click **OK**. The Library function is inserted into the logic.
Add a built-in function	a. Click Fx . The **Select Built-in Function** dialog box opens. b. From the **Filter** drop-down list, select the type of built-in function you want to add. c. From the built-in function list, select the built-in function you want. d. Click **OK**. The built-in function is inserted into the logic. For more information about built-in functions, see "Built-in functions" on page 63.
Add a condition	a. Click >= . The **Select a Condition** dialog box opens. b. Select the condition you want. c. Click **OK**. The condition is inserted into the logic.

3. If your Library function will return a value, you must have one of the following prefixes at the beginning of the code for the formula. If you combine multiple formulas into a single formula variable, each separate calculation must have its own prefix.

Type of variable	Required prefix

Static variable (non-array)	Use one of the following prefixes:
	• `Value =`
	• `Integer =`
	• `[variable name] =`
	If you are using the variable as a counter variable, do not use the `Value=` prefix or you can receive incorrect results.
	For example:
	`[variable name]=[variable name] + 1`
	A completed example might read as follows:
	`Total=Total + 1`

Array variable	Use one of the following prefixes:
	• `Value(i)=`
	• `Integer(i)=`
	• `[variable name](i)=`
	With these prefixes, `i` is equal to the number of elements you want to use.
	You can use any of the above prefixes if you are using the variable as a counter variable. The subscript i acts as a counter.
	For example:
	`Integer i`
	`Value(i) = <CreditCardAmount> + <DebitCardAmount>`
	`NEXT I`

Note: If you receive the error message `#5135, Function returns a value`, make sure you included one of the prefixes in front of the formula.

4. From the Standard toolbar, click 💾.

   The logic is saved.

For tips on creating efficient code, see "Creating Effective and Efficient Code in Exstream" on page 189.

4.2.1 Building the Library Function Logic with a Third-Party Program

If you have a program external to Exstream that you want to use to create or edit code, you can connect Exstream with the third-party program to allow you to more easily and accurately transfer code between Exstream and third-party program.

To use a third-party program:

1. If not previously set, specify the default third-party program you want to use for coding by doing the following:

   - In Design Manager, from the Menu bar, select **Tools > Options**.

     The **Options** dialog box opens.

   - In the **Data Mapping** area, in the **Text editor** box, click 📄, and then select the EXE file for the program where you want to edit the code.

   - Click **OK**.

     The **Options** dialog box closes.

   For information on setting the options for Design Manager and Designer, see *Getting Started* in the Exstream Design and Production documentation.

2. In Design Manager, from the Library, drag the Library function to the Property Panel.

3. Click ↗.

   The logic element opens in the third-party program. Any existing code for this logic element is also exported and appears in the third-party program.

4. In the third-party program, make changes as needed to the code. Do not change the name of the file. For code to import properly when you are done, you must use file name given by Exstream.

5. Save all changes.

6. In Design Manager, from the Library, drag the Library function to the Property Panel.

7. Click ⬇.

   The code, along with your changes, is imported back into Exstream.

8. From the Standard toolbar, click 💾.

   The logic is saved.

4.3 Using Library Function Arguments

Arguments are used with Library functions if the person using the Library function must enter specific information to complete calculations or actions within the Library function. For example, if you are creating a Library function to count the number of data sections across all customer output, you might want to add an optional argument where you can specify the name of a specific section you want to count.

You must add an argument for each piece of outside input you require from the Library function user. Required arguments are those entries that are mandatory in order to complete the calculation or action of the Library function. However, if you have information that will function correctly with a default if no other information is supplied, you can add an optional argument.

4.3.1 Adding Arguments to a Library Function

When you add arguments to a Library function, add all mandatory arguments before you begin to add optional arguments. After you have added an optional argument to a Library function, you can no longer add mandatory arguments to the Library function.

To add an argument to a Library function:

1. In Design Manager, from the Library, drag the Library function to the Property Panel.
2. In the **Arguments** area, click ➕.

   The **Function Argument Definition** dialog box opens.
3. In the **Name** box, enter a name for the argument. The argument is referenced in the Library function logic by this name.
4. From the **Data type** drop-down list, select the type of data accepted by this argument. Select one of the following options:
   - **String**—Text
   - **Integer**—Whole number
   - **Boolean**—True or false
   - **Floating**—Numeric value with decimals
   - **Date**—Date (with or without time of day)
   - **Currency**—Monetary amount
5. If the argument requires an array variable, select the **Array** check box.
6. If the argument is not required for the Library function or subroutine, do the following:
   a. Select the **Optional** check box.
   b. In the **Default** box, enter the default value of the argument if nothing is entered for the argument when it is defined.

> **Note:** If the **Optional** check box is selected and inactive, the last argument in the **Arguments** area was an optional argument. You cannot add mandatory arguments after an optional argument. Optional arguments must be inserted at the end of a Library function or subroutine's argument list.

7. If you add an argument that requires variable information, you must also specify the method by which the argument retrieves variable information. Select one of the following options:

To	Do this
Allow the value of the variable to continually change based on the calculations or modifications you make within the Library function	From the **Passed by** area, select the **Reference** radio button. Argument values which are passed by reference must be a variable or an array element, and the data types must match. Any calculations or modifications applied to the variable within the Library function will also affect the value of the variable after the Library function is used, if the variable is used elsewhere in your application.
Require that calculations always begin with the starting value of the variable	From the **Passed by** area, select the **Value** radio button. Any calculations or modifications applied to the variable within the Library function will not affect the value of the variable after the Library function is used.

> **Note:** If you are creating an optional argument or an argument that requires an array variable, the **Passed by** options are not available. Arrays and optional arguments are always passed by reference.

8. Click **OK**.

   The argument is added to the **Arguments** area of the Library function in the Property Panel.

9. From the Standard toolbar, click 💾.

   The Library function is saved.

4.3.2 Editing a Library Function Argument

1. In Design Manager, from the Library, drag the Library function to the Property Panel.
2. In the **Arguments** area, select the argument you want to edit.
3. Click ✏ .

   The **Function Argument Definition** dialog box opens.

4. Make changes as needed.

5. Click **OK**.

   The argument is added to the **Arguments** area of the Library function in the Property Panel.

6. From the Standard toolbar, click 💾.

   The Library function is saved.

4.3.3 Changing the Order of Arguments in a Library Function

By default, arguments appear in the order in which you created them. If needed, you can change the order of arguments for the Library functions. However, optional arguments must remain at the bottom of any argument list.

1. In Design Manager, from the Library, drag the Library function to the Property Panel.
2. In the **Arguments** area, select the argument you want to move.
3. Click ⬆ or ⬇ as needed to move the argument up and down in the list order.
4. From the Standard toolbar, click 💾.

   The Library function is saved.

4.3.4 Deleting a Library Function Argument

1. In Design Manager, from the Library, drag the Library function to the Property Panel.
2. In the **Arguments** area, select the argument you want to delete.
3. Click ⊟.

   The argument is removed from the **Argument** list.

4. From the Standard toolbar, click 💾.

   The Library function is saved.

4.4 Applying Library Functions and Subroutines to an Application

After a Library function or subroutine is created, it is possible to call the Library function or subroutine from other functions, subroutines, formulas, and rules.

To call a Library function, you must precede the Library function with `Value =`, or you receive an error message.

For example:

```
Value = Function(argument list)
```

To call subroutines, you must precede the subroutine name with `Call`.

For example:

```
Call Subroutine(argument list)
```

4.5 Testing or Troubleshooting Library Functions

When creating Library functions, you can use the code trace troubleshooting tool to make sure that your Library functions provide the results you require. Code trace records each line of source code as it is executed by the engine and, depending on your settings, returns additional information about the logic.

4.5.1 Setting a Code Trace Filter on a Library Function

You can trace code to collect information about what is happening to the logic as the engine processes your application. This information can be helpful if you need to test or troubleshoot the timing at which the code is read by the engine, or if you are receiving unexpected results. Before you can run code trace on logic, you must set a filter that specifies the amount of information you want the code trace to return on the logic you are testing.

To set a code trace filter on a Library function:

1. In Design Manager, from the Library, drag the Library function to the Property Panel.
2. From the **Code trace** drop-down list, select the code trace filter you want to use for the

Library function:

To	Do this
Write each line of the Library function to the debug file as it is executed	Select **Source Line**.
Write each line of the Library function and the names of variables to the debug file as they are used	Select **Assignment**.
Write each line of the Library function and the value of variables to the debug file as they are used	Select **All Variables**.

3. From the Standard toolbar, click 💾.
4. Run a code trace report.

   The report contains the information you requested for the logic.

For information about running a code trace report, see *Preparing Applications for Production* in the Exstream Design and Production documentation.

4.6 Using Library Functions with Effectivity

If you have licensed the Compliance Support module, you can use Library functions and subroutines with customer effectivity. With customer effectivity, you can maintain multiple versions of the same Library function or subroutine, and, when the customer output is generated, the engine selects the correct version of the Library function or subroutine to use based on the customer effectivity date.

For example, you can have a Library function to calculate the dividends for a customer in 2020 and a second version to calculate the dividends for a customer in 2021. If you package for a range of dates that includes the date approved for both versions of the Library function, then both are included in the package file. The appropriate version is picked for each customer based on the current setting for the 'SYS_CustomerEffectivity' variable.

When you are using effectivity with Library functions, the engine performs a check at run time. The different versions of the Library functions are compared to each other to make sure the argument lists match. If the arguments do not match in each Library function, you receive an error message and the Library function cannot be used.

For more information on applying effectivity and versioning to design objects, see *Designing Customer Communications* in the Exstream Design and Production documentation.

Chapter 5: Built-in functions

In contrast to Library functions, which you create, built-in functions are pre-created logic which you can modify to meet your needs. The built-in functions provided by Exstream allow you to perform routine data manipulation, such as format conversion and value reporting. For example, you can use a built-in function to extract customer information for use with local variables or to calculate new values based on customer data.

Built-in functions are available to provide a variety of actions with your data. When you add a built-in function to logic, you can locate functions easily by filtering the functions by categories based on the actions they perform or based on the specific type of data they can manipulate. The following table explains the different types of built-in functions available through Exstream.

Types of built-in functions

Type of built-in function	Description
Array	Use to manipulate or retrieve information about an array of data elements.
Data Section	Use to manipulate or retrieve information about data sections.
Date	Use to manipulate or retrieve information about dates.
General	Use to perform a variety of actions, specific to the design and production environment.
I/O	Use to set input/output information for the engine.
Live	Use to perform a variety of actions specific to LiveEditor. These functions are not discussed in this guide. For more information about built-in functions used with Live, see *Designing for LiveEditor* in the Exstream Design and Production documentation.
Math	Use to perform mathematical operations, such as adding and averaging.
String	Use to perform a variety of actions with string functions.

For more information about adding built-in functions to logic, see the chapter for the type of logic you are creating.

5.1 Array functions

Array functions let you manipulate or return information about an array of data elements. This section contains the references for all array functions.

5.1.1 Average

`Average` calculates the average value within a specified range of array elements.

Syntax

```
Average(Array[, Start, Stop])
```

Average arguments

Argument	Data type	Use	Description
`Array`	Integer, Float, or Currency	Required	The array variable containing the data you want to average
`Start`	Integer	Optional	The number of the first array element in the range you want to average The default is the first element in the array.
`Stop`	Integer	Optional	The number of the last array element in the range you want to average The default is the last element in the array.

Returns

`Average` returns the average value of the specified range of elements in the specified array. The return value is the same data type as the variable you specify in the `Array` argument.

Example

Suppose you want the average of an array named "Dependents", which contains the integers 5, 4, and 5. The syntax for `Average` might read as follows:

```
Average(Dependents[,1,3])
```

5.1.2 Contains

`Contains` locates the first element in an array that contains a specific value.

Syntax

```
Contains(Array, Target[, Start, Stop, Substring, NoCase, Sorted])
```

Contains arguments

Argument	Data type	Use	Description
`Array`	Any	Required	The array variable you want to search
`Target`	Same as `Array`	Required	The value for which you want to search
`Start`	Integer	Optional	The number of the first array element in the range you want to search The default is the first element in the array.

Contains arguments, continued

Argument	Data type	Use	Description
`Stop`	Integer	Optional	The number of the last array element in the range you want to search The default is the last element in the array.
`Substring`	Boolean	Optional	The Boolean value that indicates whether you want to also search substrings for the `Target` value Enter one of the following Boolean conditions: • `TRUE`—The target string can match only part of the value of the array element. • `FALSE`—The target string must exactly match the value of the array element. The default is `FALSE`.
`NoCase`	Boolean	Optional	The Boolean value that indicates whether you want to consider case when searching for the `Target` value Enter one of the following Boolean conditions: • `TRUE`—Strings are compared ignoring case. • `FALSE`—Strings must match exactly, including case. The default is `FALSE`.
`Sorted`	Boolean	Optional	The Boolean value that indicates whether the array is sorted. This determines whether the search is binary or linear. A binary search on data that is already sorted is faster than a linear search, particularly on arrays with ten or more elements. Enter one of the following Boolean conditions: • `TRUE`—The array content is sorted and you want to conduct a binary search on the array. If you specify `TRUE` and the array is not sorted, unexpected and inaccurate results may occur. • `FALSE`—The array content is not sorted and you want to conduct a linear search on the array. The default is `FALSE`.

Returns

`Contains` returns the index of the first array element that matches the value of the `Target` argument. If there is no match of the `Target` within the array, the function returns a zero (`0`). The return value is in the integer data type.

Example

Suppose you want to send a coupon for a free gift to any customer who has subscribed to your wireless program. Your built-in function might read as follows:

```
Contains(CustomerSubscriptions, Wireless, FALSE, TRUE, FALSE)
```

Since you know that any customer who has subscribed to your wireless program receives a non-zero value for this built-in function, you could use this function in combination with a rule to include the coupon if the value of `Contains` is greater than zero.

5.1.3 ContainsNot

`ContainsNot` locates the first element in an array that does not contain a specific value.

Syntax

```
ContainsNot(Array, Target[, Start, Stop, Substring, NoCase])
```

ContainsNot arguments

Argument	Data type	Use	Description
`Array`	Any	Required	The array variable you want to search
`Target`	Same as `Array`	Required	The value for which you want to search
`Start`	Integer	Optional	The number of the first array element in the range you want to search The default is the first element in the array.
`Stop`	Integer	Optional	The number of the last array element in the range you want to search The default is the last element in the array.
`Substring`	Boolean	Optional	The Boolean value that indicates whether you want to also search substrings for the `Target` value Enter one of the following Boolean conditions: • `TRUE`—The target string can match only part of the value of the array element. • `FALSE`—The target string must exactly match the value of the array element. The default is `FALSE`.

ContainsNot arguments, continued

Argument	Data type	Use	Description
`NoCase`	Boolean	Optional	The Boolean value that indicates whether you want to consider case when searching for the `Target` value Enter one of the following Boolean conditions: • `TRUE`—Strings are compared ignoring case. • `FALSE`—Strings must match exactly, including case. The default is `FALSE`.

Returns

`ContainsNot` returns the index of the first element that does not match the value of the `Target` argument. If all the elements of the array match value of the `Target` argument, this function returns a `0` (zero) to indicate failure. The return value is in the integer data type.

5.1.4 Count

`Count` counts the number of elements in an array.

5.1.5 Syntax

`Count(Array)`

Count argument

Argument	Data type	Use	Description
`Array`	Any	Required	The array variable to count

Returns

`Count` returns the number of elements in the array you specified in the `Array` argument. The return value is in the integer data type.

Example

Suppose that you have an array that contains one element for each year that a customer has subscribed to your newsletter. If you want to find out how many years the customer has received your newsletter, you could use `Count` to count the number of elements in the array.

5.1.6 Insert

`Insert` inserts an array element at a specified location in an array.

Syntax

```
Insert(Array, Index, Target)
```

Insert arguments

Argument	Data type	Use	Description
`Array`	Any	Required	The array variable to which you want to add an element
`Index`	Integer	Required	The index of the array where `Target` is to be inserted
`Target`	Same as `Array`	Required	The value of the element you want to insert. The data type of the `Target` argument is automatically converted to match the data type of the `Array` argument.

Returns

`Insert` does not return a value.

5.1.7 Join

`Join` concatenates all the elements of a string array together into a single string, optionally separating each element with a delimiter.

Syntax

```
Join(Array[, "Delimiter"])
```

Join arguments

Argument	Data type	Use	Description
`Array`	String	Required	The array variable containing the elements you want to concatenate You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.

Join arguments, continued

Argument	Data type	Use	Description
`Delimiter`	String	Optional	The character you want to use as a delimiter **Tip:** If you do not want spaces between the array elements, use double quotation marks as the delimiter. You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable. The default is a space.

Returns

`Join` returns the concatenated string. The return value is in the string data type.

5.1.8 Max

`Max` identifies the maximum value in a specified array.

Syntax

```
Max(Array[, Start, Stop])
```

Max arguments

Argument	Data type	Use	Description
`Array`	Any	Required	The array variable you want to search
`Start`	Integer	Optional	The number of the first array element in the range you want to search The default is the first element in the array.
`Stop`	Integer	Optional	The number of the last array element in the range you want to search The default is the last element in the array.

Returns

`Max` returns the maximum value in the array specified in the `Array` argument. The return value is in the same data type as the array specified in the `Array` argument.

5.1.9 Min

`Min` identifies the minimum value in a specified array.

Syntax

```
Min(Array[, Start, Stop])
```

Min arguments

Argument	Data type	Use	Description
`Array`	Any	Required	The array variable you want to search
`Start`	Integer	Optional	The number of the first array element in the range you want to search The default is the first element in the array.
`Stop`	Integer	Optional	The number of the last array element in the range you want to search The default is the last element in the array.

Returns

`Min` returns the minimum value in the array specified in the `Array` argument. The return value is in the same data type as the array specified in the `Array` argument.

5.1.10 Remove

`Remove` removes a specific element from an array and adjusts the location of any succeeding elements accordingly.

Syntax

```
Remove(Array, Index)
```

Remove arguments

Argument	Data type	Use	Description
`Array`	Any	Required	The array variable that contains the element you want to remove
`Index`	Integer	Required	The number of the array element you want to remove from the array. Elements following the removed element are shifted to fill the gap.

Returns

`Remove` does not return a value.

5.1.11 Resize

`Resize` changes the size of a global array, removing trailing elements if necessary.

Syntax

```
Resize(Array, Index)
```

Resize arguments

Argument	Data type	Use	Description
`Array`	Any	Required	The array variable you want to resize. You must use a global array.
`Index`	Integer	Required	The number of the array element after which you want to remove all trailing elements

Returns

`Resize` does not return a value.

5.1.12 Sort

`Sort` sorts a key array and up to nine additional arrays into the same order as the key array. This function uses a stable sort, because any duplicate values are maintained in their original order.

You must use at least three arguments with this function.

Syntax

```
Sort(KeyArray, "A/D", Array1[, Array2..., Array9])
```

Sort arguments

Argument	Data type	Use	Description
`KeyArray`	Any	Required	The array variable you want to sort. This array drives the sorting of any subsequent arrays.

Sort arguments, continued

Argument	Data type	Use	Description
`A/D`	String	Required	The order in which you want to sort the elements of the array Enter one of the following values: • `A`—Ascending order • `D`—Descending order You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`Array1...Array9`	Any	Required	The arrays you want to sort with the key array. You can include up to nine arrays. Each array you use must contain the same number of elements as the array you specify for the `KeyArray` argument. You must use global arrays. **Note:** If the arrays specified by the `Array1...Array9` arguments do not contain the same number of elements as `KeyArray`, you receive an error message and no sorting is performed. The function returns `0` instead of the number of elements in `KeyArray`.

Returns

`Sort` returns the number of elements in the array you enter for the `KeyArray` argument. The return value is in the integer data type. If you receive an error, the return value is `0`.

5.1.13 SortMulti

`SortMulti` sorts data using one or more arrays as the sort keys. This function uses a stable sort, because any duplicate values are maintained in their original order.

> **Caution:** You receive an error message if you use a local array with `SortMulti`. You must use global arrays in this function.

Syntax

```
SortMulti(Array1, "A/D"[, Array2, "A/D"...])
```

SortMulti arguments

Argument	Data type	Use	Description
`Array1`	Any	Required	The first array variable to use as a key for sorting. You must use a global array.
`A/D`	String	Required	The order in which you want to sort the elements of the array Enter one of the following values: • `A`—Ascending order • `D`—Descending order You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`Array2, A/D`	Any	Optional	Additional array(s) and sorting specifications to use for sorting You can specify up to 127 arrays, depending on your available memory resources. Arrays are used as sort keys in the order they are encountered. Each array you use must contain the same number of elements as the array you specify for the `KeyArray` argument. You must use global arrays.

Returns

`SortMulti` returns the number of data elements sorted. The return value is in the integer data type. If you receive an error, the return value is `0`.

5.1.14 SortRange

`SortRange` sorts a specific range of elements in one or more arrays using one or more arrays as the sort keys. This function uses a stable sort, because any duplicate values are maintained in their original order.

Syntax

```
SortRange(Start, End, Array1, A/D[, Array2...])
```

SortRange arguments

Argument	Data type	Use	Description
`Start`	Integer	Required	The number of the first array element in the range you want to sort Use `1` to indicate the first element in the array. **Caution:** Use caution when specifying the `Start` and `End` arguments. You receive an error message if the arrays do not have at least the number of elements specified by `End`, or if `End` is less than or equal to `Start`.
`End`	Integer	Required	The number of the last array element in the range you want to sort Use `-1` to indicate the last element in the array. **Caution:** Use caution when specifying the `Start` and `End` arguments. You receive an error message if the arrays do not have at least the number of elements specified by `End`, or if `End` is less than or equal to `Start`.
`Array1`	Any	Required	The first array variable to use as a key for sorting. You must use a global array.
`A/D`	Boolean	Required	The order in which you want to sort the elements of the array Enter one of the following values: • A—Ascending order • D—Descending order
`Array2, A/D`	Any	Optional	Additional array(s) to use for sorting

Returns

`SortRange` returns the number of data elements sorted. The return value is in the integer data type. If you receive an error, the return value is `0`.

Example

Suppose you know that a particular ZIP Code starts at element `25` and ends at `125`. You can sort these elements by the customers' last name, then first name.

5.1.15 Split

`Split` divides a delimited string into an array of substrings.

> **Tip:** If you have a string that is not delimited and you want to parse each character of the string into a separate array element, you should use a `FOR loop` instead of the `Split` function.

Syntax

```
Split(Array, "String"[, "Delimiter", Limit])
```

Split arguments

Argument	Data type	Use	Description
`Array`	String	Required	The array variable in which you want to place the resulting sub-strings You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`String`	String	Required	The source string you want to split into substrings **Note:** If the maximum number of array elements to create is exceeded, the remainder of the string is placed in the final element. You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`Delimiter`	String	Optional	The character you want to use as a delimiter You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable. The default is a space. If you enter a null value (`""`) or an invalid delimiter, the entire string is moved into the first array element.
`Limit`	Integer	Optional	The maximum number of elements that can be created. By default, there is no limit.

Returns

`Split` returns the number of substrings created. The return value is in the integer data type.

5.1.16 Sum

Sum calculates the total of all the values in a specific range of elements within a numeric array.

Syntax

```
Sum(Array[, Start, Stop])
```

Sum arguments

Argument	Data type	Use	Description
Array	Integer, Float, or Currency	Required	The array variable containing the numeric data to total
Start	Integer	Optional	The number of the first array element in the range you want to add The default is the first element in the array.
Stop	Integer	Optional	The number of the last array element in the range you want to add The default is the last element in the array.

Returns

Sum returns the total of all the values in the specified range of elements within the array specified in the Array argument. The return value is in the same data type as the array specified in the Array argument.

5.1.17 TrimArray

TrimArray removes the trailing elements you specify from an array.

Syntax

```
TrimArray(Array[, Value])
```

TrimArray arguments

Argument	Data type	Use	Description
Array	Any	Required	The array variable you want to trim. You must use a global array variable.
Value	Same as Array	Optional	The default is an empty string or 0, depending on the data type of the Array argument.

Returns

TrimArray returns the count of the remaining elements in the array you specified in the Array argument. The return value is in the integer format.

5.1.18 WhenAverage

WhenAverage allows you to use a match array to identify specific elements, and then use the specified elements to identify and average the matching elements in a main array.

This function uses two arrays. The first array, Array, contains the values you want to use to calculate an average. The second array, MatchArray, is compared with a target. The function locates the elements in MatchArray that contain the Target value and then locates and averages the corresponding elements in Array.

Syntax

```
WhenAverage(Array, MatchArray, Target[, SubString, NoCase])
```

WhenAverage arguments

Argument	Data type	Use	Description
Array	Integer, Float, or Currency	Required	The array variable in which you want to search
MatchArray	Any	Required	The array variable within which you want to search for matches to the Target argument If the array specified for the MatchArray argument contains more elements than the array specified for the Array argument, only the number of elements in Array are searched in MatchArray. The remaining elements are ignored.
Target	Same as MatchArray	Required	The value for which you want to search
Substring	Boolean	Optional	The Boolean value that indicates whether you want to also search substrings for the **Target** value Enter one of the following Boolean conditions: • TRUE—The target string can match only part of the value of the array element. • FALSE—The target string must exactly match the value of the array element. The default is FALSE.

WhenAverage arguments, continued

Argument	Data type	Use	Description
NoCase	Boolean	Optional	The Boolean value that indicates whether you want to consider case when searching for the Target value • TRUE—Strings are compared ignoring case. • FALSE—Strings must match exactly, including case. The default is FALSE.

Returns

WhenAverage returns the average of the matching elements in the array specified for the Array argument. The return value is in the same data type as the array you specified for the Array argument.

Example

Suppose that the table below represents the two arrays.

Array	MatchArray
10	!
20	@
30	!
40	@
50	!

If you specify Target to be !, three matches are found in MatchArray and the corresponding elements in Array (10, 30, and 50) are averaged. This scenario would result in an average value of 30.

5.1.19 WhenCount

WhenCount counts a subset of items in an array triggered by a value of another array. It uses two arrays. The first array, Array, controls the number of elements searched in MatchArray. The second array, MatchArray, is searched for a match to Target.

Syntax

```
WhenCount(Array, MatchArray, Target[, SubString, NoCase])
```

WhenCount arguments

Argument	Data type	Use	Description
`Array`	Any	Required	The array variable containing the elements to be counted
`MatchArray`	Any	Required	The array variable within which you want to search for matches to the `Target` argument If the array specified for the `MatchArray` argument contains more elements than the array specified for the `Array` argument, only the number of elements in `Array` are searched in `MatchArray`. The remaining elements are ignored.
`Target`	Same as `MatchArray`	Required	The value for which you want to search
`Substring`	Boolean	Optional	The Boolean value that indicates whether you want to also search substrings for the `Target` value Enter one of the following Boolean conditions: • `TRUE`—The target string can match only part of the value of the array element. • `FALSE`—The target string must exactly match the value of the array element. The default is `FALSE`.
`NoCase`	Boolean	Optional	The Boolean value that indicates whether you want to consider case when searching for the `Target` value Enter one of the following Boolean conditions: • `TRUE`—Strings are compared ignoring case. • `FALSE`—Strings must match exactly, including case. The default is `FALSE`.

Returns

`WhenCount` returns the count of the number of elements that match `Target`. The return value is in the integer data type.

Example

Suppose that the table below represents the two arrays.

Array	MatchArray
`10`	`!`

Array	MatchArray
`20`	`@`
`30`	`!`
`40`	`@`
`50`	`!`

If you specify `Target` to be `!`, three matches are found and `3` is returned.

5.1.20 WhenSum

`WhenSum` allows you to use a match array to identify specific elements, and then use the specified elements to identify and add matching elements in a main array.

This function uses two arrays. The first array, `Array`, contains the values that are used to perform the computation. The second array, `MatchArray`, is searched for a match to `Target`. The elements that match the `Target` are stored and the data contained in the corresponding elements of `Array` are added together.

Syntax

```
WhenSum(Array, MatchArray, Target, SubString, NoCase)
```

WhenSum arguments

Argument	Data type	Use	Description
`Array`	Integer, Float, or Currency	Required	The array variable containing the elements to be added
`MatchArray`	Any	Required	The array variable within which you want to search for matches to the `Target` argument If the array specified for the `MatchArray` argument contains more elements than the array specified for the `Array` argument, only the number of elements in `Array` are searched in `MatchArray`. The remaining elements are ignored.
`Target`	Same as `MatchArray`	Required	The value for which you want to search

WhenSum arguments, continued

Argument	Data type	Use	Description
`Substring`	Boolean	Optional	The Boolean value that indicates whether you want to also search substrings for the `Target` value Enter one of the following Boolean conditions: • `TRUE`—The target string can match only part of the value of the array element. • `FALSE`—The target string must exactly match the value of the array element. The default is `FALSE`.
`NoCase`	Boolean	Optional	The Boolean value that indicates whether you want to consider case when searching for the `Target` value Enter one of the following Boolean conditions: • `TRUE`—Strings are compared ignoring case. • `FALSE`—Strings must match exactly, including case. The default is `FALSE`.

Returns

`WhenSum` returns the sum of the matching elements in the array specified for the `Array` argument. The return value is in the same data type as the array you specified for the `Array` argument.

Example

Suppose that the table below represents the two arrays.

Array	MatchArray
10	!
20	@
30	!
40	@
50	!

If you specify `Target` to be `!`, three matches are found the sum of the corresponding elements is returned:

```
10 + 30 + 50 = 90
```

5.2 Customizing built-in function arguments

After you select the function you want to use, sample syntax for the function is placed in the code. The standard syntax for a built-in function is similar to the following example:

```
FUNCTION (arg1, [arg2, arg3, arg4])
```

In the syntax for the built-in function, there are commonly arguments (above, represented by `arg1` through `arg4`) which require information from your customer data or require you to set your preference for how the built-in function functions. You must add information, as needed, for the arguments of the built-in function. When you review the syntax of a function, you might encounter the following types of arguments:

Types of arguments used with built-in functions

Type of argument	Description
Required arguments	Required arguments appear inside the brackets(`()`) but outside of the square brackets (`[]`). In the above example, `arg1` is required.

Types of arguments used with built-in functions, continued

Type of argument	Description

Optional arguments	Optional arguments are contained within square brackets (`[]`). In the above example, `arg2`, `arg3`, and `arg4` are optional arguments. Optional arguments have a default value. If you do not include a specific value for an optional argument, the default is used to fulfill the argument. This lets you specify only the arguments you need. Depending on the optional arguments you use, you must do one of the following to complete the built-in function:
	• **If no optional arguments are used**—If you choose not to use any optional arguments, the optional arguments can be left blank or removed from the code for the function. If you do not need to use any of the optional arguments, the sample function might look similar to the following:
	`FUNCTION (arg1, , , ) FUNCTION (arg1, , , )`
	or
	`FUNCTION (arg1)`
	• **If some optional arguments are used**—If you use an optional argument that is preceded by optional arguments you are not using, you must leave the optional arguments you don't use blank instead of removing them from the code.
	  • If you want to use only `arg1` and `arg4`, the sample function might look similar to the following:
	  `FUNCTION (arg1, , , arg4)`
	  • If you remove the optional arguments that precede an argument you want to use, you must leave the argument blank and leave all necessary commas in the built-in function syntax. If you remove the arguments and commas, you change the category to which the argument information is applied. This can give you unexpected results. For example, suppose you only wanted to use arg1 and arg4. If you removed the other arguments, the sample function might look similar to the following:
	  `FUNCTION (arg1, arg4)`
	  However, the function above reads the information you included for `arg4` as if it were information for `arg2`, since the information is in the incorrect position in the function.

Arguments that use array variables	If the variable you enter for a value in a built-in function is an array, you must specify the number of the array element you want to use in brackets (`()`) after the variable. In the above example, if you wanted `arg1` to use the first element in an array, your function might look similar to the following: `FUNCTION (arg1(1), arg2, arg3, arg4)`

5.3 Data section functions

Data section functions let you manipulate or retrieve information about data sections. This section contains the references for all data section functions.

5.3.1 AddDataSection

`AddDataSection` adds a new data section to a specific location in your data.

To use the `AddDataSection` function, you must meet the following requirements:

- The input data file must include a layout with the named data section.
- The AGGREGATE_DATA switch must be in the control file (except when DLF is the only output type).

For more information about the AGGREGATE_DATA switch, see *Switch Reference* in the Exstream Design and Production documentation.

Syntax

```
AddDataSection("DataSectionName", SiblingInstance, "ParentName",
ParentInstance, Before)
```

AddDataSection arguments

Argument	Data type	Use	Description
`DataSectionName`	String	Required	The name you want to apply to the data section you are adding You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`SiblingInstance`	Integer	Required	The number of the sibling data section relative to where you want to add the new data section • If the value you enter is less than or equal to `1`, the new section is added relative to the first sibling data section. • If the value is equal to or greater than the number of siblings, the new data section is added relative to the last sibling data section. • The value you enter for the `Before` argument determines whether the new data section is added before or after the first or last sibling data section.
`ParentName`	String	Required	Do one of the following to identify the level of data to which you want to add the data section: • If you want to make the added section a child, enter the name of the parent section to which you want to add a data section. • If you want to add the data section to the first-level children of the customer data, leave the argument blank. You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`ParentInstance`	Integer	Required	The number of the specific instance of the parent section to which you want to add a data section. If you are adding the data section to the first-level children of the customer data, leave this argument blank. This argument is required only if the sibling data section is not the highest level of data.
`Before`	Boolean	Optional	The Boolean value that indicates whether you want to place the data section you add before the `SiblingInstance` data section Enter one of the following Boolean conditions: • `TRUE`—The added data section is placed before the sibling data section specified in the `SiblingInstance` argument. • `FALSE`—The added data section is placed after the sibling data section specified in the `SiblingInstance` argument. The default is `FALSE`.

Returns

`AddDataSection` returns one of the following status codes.

AddDataSection status codes

Status code	Description
0	Success
25	No data section with the specified parent name was found.
26	No data section with the specified parent instance was found.
27	Incorrect function timing. Function must be called during the data aggregation event.
29	Invalid structure. No layout could be found for the specified data section name.

The return value is in the integer data type.

Example

Suppose you have the following data structure:

```
"Dependent"
   SSN: 12356789
   DOB: 02271987
   "Category"
      Name: Son
"Dependent"
   SSN: 234567890
   DOB: 05051991
   "Category"
      Name: Daughter
```

Also suppose you need to add another dependent category between the two existing instances. If you want to add a peer section named `"Dependent"` in between the two existing instances, use the following syntax:

```
AddDataSection("Dependent", 1, "", 0, FALSE)
```

The resulting data structure would read as follows:

```
"Dependent"
   SSN: 12356789
   DOB: 02271987
   "Category"
      Name: Son
"Dependent"
"Dependent"
   SSN: 234567890
   DOB: 05051991
   "Category"
      Name: Daughter
```

In the resulting data structure, if you wanted to append a child section named `"Category"` to the second instance of `"Dependent"`, then you can use the following syntax:

```
AddDataSection("Category", 0, "Dependent", 2, FALSE)
```

The resulting data structure would read as follows:

```
"Dependent"
   SSN: 12356789
   DOB: 02271987
   "Category"
      Name: Son
"Dependent"
   "Category"
"Dependent"
   SSN: 234567890
   DOB: 05051991
   "Category"
      Name: Daughter
```

5.3.2 AddRecipient

`AddRecipient` adds recipient copies of a document set. The `AddRecipient` function is executed during data aggregation engine timing to add the specified number of copies of the current customer's documents to the output using the specified recipient profile. You can also include this function in a Live document to add recipient copies to the DLF file during a LiveEditor session. For example, if the function is executed in Customer 4 using the "courtesy copy" recipient profile, a recipient copy of Customer 4's documents is added to the output or to the DLF file using the settings from the "courtesy copy" profile. The copy of the documents is added after the last page of the last document in the original customer.

For more information about adding recipient copies, see *Designing Customer Communications* in the Exstream Design and Production documentation.

For more information about adding recipient copies in Exstream Live, see *Designing for LiveEditor* in the Exstream Design and Production documentation.

Syntax

```
AddRecipient([RecipientProfileName, NumCopies])
```

AddRecipient arguments

Argument	Data type	Use	Description
`RecipientProfileName`	String	Optional	The recipient profile to use for the recipient copy You must enter the name of the recipient profile in quotation marks. The default is the first recipient profile listed for the application.
`NumCopies`	Integer	Optional	The number of copies to add using the selected recipient profile The default and the minimum value is `1`.

Returns

`AddRecipient` returns one of the following status codes:

AddRecipient returns

Status code	Description
`0`	Success
`21`	No customer is selected.
`27`	Incorrect function timing. Function must be called during the data aggregation event.

The return value is in the integer data type.

5.3.3 CountDataSection

`CountDataSection` counts the number of data sections in the current aggregated customer data or counts the number of times a specified data section occurs within the current aggregated customer data.

To use the `CountDataSection` function, the AGGREGATE_DATA switch must be in the control file (except when DLF is the only output type).

For more information about the AGGREGATE_DATA switch, see *Switch Reference* in the Exstream Design and Production documentation.

Syntax

```
CountDataSection("DataSectionName", InstanceNumber)
```

CountDataSection arguments

Argument	Data type	Use	Description
`DataSectionName`	String	Optional	The name of the data section you want to include in the count You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`InstanceNumber`	Integer	Optional	The instance of the specified data section or data index from which the specified data section should be included in the count

Returns

`CountDataSection` returns a count of the specified data sections. Based on your use of argument combinations, the following results are possible:

Results of CountDataSection argument combinations

DataSectionName argument value	InstanceNumberOrIndex argument value	Possible results
Non-blank	Greater than zero	A specific instance of a specific data section is included in the count.
Non-blank	Zero or blank	All instances of a specific data section are included in the count.
Blank	Greater than zero	The data section at the data index is included in the count, and the data section name is ignored.
Blank	Blank	All section data is included in the count.

The return value is in the integer data type.

5.3.4 DeleteDataSection

`DeleteDataSection` deletes one or all occurrences of the specified data section in the current aggregated customer data. Depending on your settings, you can also either delete all section data or delete the data section that occurs at a specific index.

> **Note:** If you delete a parent data section, the children are also deleted.

To use the `DeleteDataSection` function, the AGGREGATE_DATA switch must be in the control file (except when DLF is the only output type).

For more information about the AGGREGATE_DATA switch, see *Switch Reference* in the Exstream Design and Production documentation.

Syntax

```
DeleteDataSection("DataSectionName", InstanceNumberOrIndex)
```

DeleteDataSection arguments

Argument	Data type	Use	Description
`DataSectionName`	String	Optional	The name of the data section you want to delete You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`InstanceNumberOrIndex`	Integer	Optional	The instance of the specified data section or data index from which the specified data section should be deleted

The return value is in the integer data type.

Returns

`DeleteDataSection` returns the number of sections deleted. Based on your use of argument combinations, the following results are possible:

Results of DeleteDataSection argument combinations

DataSectionName argument value	InstanceNumberOrIndex argument value	Possible results
Non-blank	Greater than zero	A specific instance of a specific data section is deleted.
Non-blank	Zero or blank	All instances of a specific data section are deleted.
Blank	Greater than zero	The data section at the data index is deleted, and the data section name is ignored.
Blank	Blank	All section data is deleted.

The return value is in the integer data type.

5.3.5 DuplicateDataSection

`DuplicateDataSection` creates a copy of the currently selected data section and, optionally, all its variable values.

> **Note:** If you duplicate a parent data section, the children are also duplicated.

To use the `DuplicateDataSection` function, you must meet the following requirements:

- The `SelectDataSection` function must be included before the `DuplicateDataSection` function in the logic.
- The AGGREGATE_DATA switch must be in the control file (except when DLF is the only output type).

For more information about the AGGREGATE_DATA switch, see *Switch Reference* in the Exstream Design and Production documentation.

Syntax

```
DuplicateDataSection(CopyTheData, Before, SelectNewDataSection)
```

DuplicateDataSection arguments

Argument	Data type	Use	Description
`CopyTheData`	Boolean	Optional	The Boolean value that indicates whether you want the copy to retain the same values as the original Enter one of the following Boolean conditions: • `TRUE`—Variables in the copied section retain their values. • `FALSE`—Variables in the copied section have no values. The default is `FALSE`.
`Before`	Boolean	Optional	The Boolean value that indicates whether you want the copy to be placed before the original data section Enter one of the following Boolean conditions: • `TRUE`—The copy of the data section is placed before the original data section. • `FALSE`—The copy of the data section is placed after the original data section. The default is `FALSE`.
`SelectNewDataSection`	Boolean	Optional	The Boolean value that indicates whether you want select the copied data section when the function is complete Enter one of the following Boolean conditions: • `TRUE`—The copy of the data section is selected. • `FALSE`—The original data section is selected The default is `FALSE`.

Returns

`DuplicateDataSection` returns one of the following status codes:

DuplicateDataSection status codes

Status code	Description
`0`	Success
`27`	Incorrect function timing. Function must be called during the data aggregation event.
`28`	No data section selected. You must use the `SelectDataSection` function first.

The return value is in the integer data type.

Example

For example, suppose you have the following data structure:

```
"Dependent"
   SSN: 12356789
   DOB: 02271987
   "Category"
      Name: Son
```

If you want to duplicate this section and keep it selected afterward, you can use the following syntax:

```
SelectDataSection("Dependent", 1)
DuplicateDataSection(TRUE, FALSE, FALSE)
```

The resulting data structure would read as follows:

```
"Dependent"
   SSN: 12356789
   DOB: 02271987
   "Category"
      Name: Son
"Dependent"
   SSN: 12356789
   DOB: 02271987
```

In the same existing data structure, if you want to duplicate the section but select the copy afterward, you can use the following syntax:

```
SelectDataSection("Dependent", 1)
DuplicateDataSection(TRUE, FALSE, TRUE)
```

5.3.6 MergeDataSection

`MergeDataSection` appends the array variables of the source data section to those of the target data section. Scalar variables are not affected. Child data sections are made children of the target data section. After the merge, variables for the specified data section are set to the values of the target instance number.

> **Note:** If you merge a parent data section, the children are also merged.

In order for the section data to merge the content, the data in each section must be balanced. Compare the two following examples to see the difference between balanced data sections and unbalanced data sections.

- **Balanced data sections**

  ```
  <Policy>Deductible</Policy>
       <Plan>A</Plan>
       <Value>500</Value>
  <Policy>Deductible</Policy>
       <Plan>B</Plan>
       <Value>500</Value>
  ```

- **Unbalanced data sections**

  ```
  <Policy>Deductible</Policy>
       <Plan>A</Plan>
       <Value>500</Value>
            <Info>Out of Pocket Max</Info>
            <Value>2500</Value>
  <Policy>Deductible</Policy>
       <Plan>B</Plan>
       <Value>500</Value>
  ```

To use the `MergeDataSection` function, the AGGREGATE_DATA switch must be in the control file (except when DLF is the only output type).

For more information about the AGGREGATE_DATA switch, see *Switch Reference* in the Exstream Design and Production documentation.

Syntax

```
MergeDataSection("DataSectionName", TargetInstanceNumber, InstanceNumber)
```

MergeDataSection arguments

Argument	Data type	Use	Description
`DataSectionName`	String	Required	The name of the data section you want to merge You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`TargetInstanceNumber`	Integer	Required	The number for the instance of the data section that is the destination of the merge
`InstanceNumber`	Integer	Required	The instance of the data section you want to merge with the data section specified in the `TargetInstanceNumber` argument

Returns

`MergeDataSection` returns one of the following status codes:

MergeDataSection status codes

Status code	Description
`0`	Success. This code is also returned if the target and source index are the same and no change occurs.
`25`	No data section with the specified name was found.
`26`	No data section with the specified instance was found.
`27`	Incorrect function timing. Function must be called during the data aggregation event.

The return value is in the integer data type.

Example

Suppose you have the following data structure:

```
"Deductible"
   Plan: A
   Value: 500
"Deductible"
   Plan: C
   Value: 1000
"Deductible"
   Plan: B
   Value: 500
```

If you want to merge the two instances with matching variable values to eliminate repetition, you can use the following syntax:

```
MergeDataSection("Dependent", 1, 3)
```

The resulting data structure would read as follows:

```
"Deductible"
   Plan: A, B
   Value: 500
"Deductible"
   Plan: C
   Value: 1000
```

However, using the same original data structure, suppose you use the following syntax:

```
MergeDataSection("Dependent", 3, 1)
```

The resulting data structure would read as follows:

```
"Deductible"
   Plan: C
   Value: 1000
"Deductible"
   Plan: B, A
   Value: 500
```

5.3.7 MoveDataSection

`MoveDataSection` moves the selected data section to a specified location. This function returns a status code.

> **Note:** If you move a parent data section, the children are also moved. You cannot move a parent data section to a current child data section.

To use the `MoveDataSection` function, you must meet the following requirements:

- The `SelectDataSection` function must be included before the `MoveDataSection` function in the logic.
- The AGGREGATE_DATA switch must be in the control file (except when DLF is the only output type).

For more information about the AGGREGATE_DATA switch, see *Switch Reference* in the Exstream Design and Production documentation.

Syntax

```
MoveDataSection(SiblingInstance, "ParentName", ParentInstance, Before)
```

MoveDataSection arguments

Argument	Data type	Use	Description
`SiblingInstance`	Integer	Required	The target sibling data section relative to where you want to move the selected data section • If the value is less than or equal to `1`, the function moves the selected data section relative to the first sibling data section, based on the value of the `Before` argument. • If the value is equal to or greater than the number of siblings, the function moves the selected data section after the final data section. Moved data sections are appended after the children of the data section to which they are moved.
`ParentName`	String	Required	The name of the parent data section that contains the sibling data section relative to where you want to move the selected data section You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable. This argument is required only if the selected data section is not the highest level of data.
`ParentInstance`	Integer	Required	If you use the `ParentName` argument, the specific instance of the parent data section that contains the sibling data section relative to where you want to move the selected data section • If the value is less than or equal to `1`, the function moves the selected data section relative to the first parent section, based on the value of the `Before` argument. • If the value is equal to or greater than the number of data sections available, the function moves the selected data section after the final data section. Moved data sections are appended after the children of the data section to which they are moved. This argument is required only if the selected data section is not the highest level of data.
`Before`	Boolean	Optional	The Boolean value that indicates whether you want the moved data section to be placed before the sibling data section specified in the `SiblingInstance` argument Enter one of the following Boolean conditions: • `TRUE`—The selected data section is placed before the sibling data section specified in the `SiblingInstance` argument. • `FALSE`—The selected data section is placed after the sibling data section specified in the `SiblingInstance` argument. The default is `FALSE`.

Returns

`MoveDataSection` returns one of the following status codes:

MoveDataSection status codes

Status code	Description
`0`	Success
`25`	No data section with the specified parent name was found.
`26`	No data section with the specified parent instance was found.
`27`	Incorrect function timing. Function must be called during the data aggregation event.
`28`	No data section is selected. You must use the `SelectDataSection` function first.
`29`	Invalid structure. The specified parent section is a child of the data section being moved.

The return value is in the integer data type.

Example

Suppose you have the following data structure:

```
"Dependent"
   SSN: 12356789
   DOB: 02271987
   "Category"
      Name: Son
"Dependent"
   SSN: 234567890
   DOB: 05051991
   "Category"
      Name: Daughter
```

If you want to move the second instance of the section named `"Dependent"` before the first instance, you can use the following syntax:

```
SelectDataSection("Dependent", 2)
MoveDataSection(1, "", "", TRUE)
```

The resulting data structure would read as follows:

```
"Dependent"
   SSN: 234567890
   DOB: 05051991
   "Category"
      Name: Daughter
"Dependent"
   SSN: 12356789
   DOB: 02271987
   "Category"
      Name: Son
```

5.3.8 SelectDataSection

`SelectDataSection` selects the variable values for the specified data section. This function returns a status code.

To use the `SelectDataSection` function, the AGGREGATE_DATA switch must be in the control file (except when DLF is the only output type).

For more information about the AGGREGATE_DATA switch, see *Switch Reference* in the Exstream Design and Production documentation.

Syntax

```
SelectDataSection("DataSectionName", InstanceNumberOrIndex, ChildNumber)
```

SelectDataSection arguments

Argument	Data type	Use	Description
`DataSectionName`	String	Required	The name of the data section you want to select You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`InstanceNumberOrIndex`	Integer	Required	The instance or data index of the data section you want to select
`ChildNumber`	Integer	Optional	The number of the child data index you want to select The default is `0`.

Returns

`SelectDataSection` returns one of the following status codes:

SelectDataSection status codes

Status code	Description
`0`	Success
`25`	No data section with the specified name was found.
`26`	No data section with the specified instance or index was found.
`27`	Incorrect function timing. Function must be called during the data aggregation event.

The return value is in the integer data type.

5.4 Date functions

Date functions let you manipulate or retrieve information about dates. This section contains the references for all date functions.

5.4.1 Date

`Date` retrieves the current system date.

Syntax

```
Date()
```

> **Caution:** `Date` has no arguments. If you enter an argument, you receive an error message.

Returns

`Date` returns the current system date. The return value is in the date data type.

5.4.2 DateAdd

`DateAdd` adds a specific interval of time to a date. This can be used to calculate a future date (by adding a positive number of intervals) or used to calculate a past date (by adding a negative number of intervals).

Syntax

```
DateAdd("Interval", Number of Periods, Original Date)
```

DateAdd arguments

Argument	Data type	Use	Description
`Interval`	String	Required	The string that represents the interval of time you want to use to measure the amount of time to add to the original date Enter one of the following codes: • `yyyy`—Year • `q`—Quarter • `m`—Month • `y`—Day of year • `d`—Day • `w`—Weekday • `ww`—Week • `h`—Hour • `n`—Minute • `s`—Second You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`Number of Periods`	Integer	Required	The number of times you want to add the specified interval to the original date. For example, if you want to add three years to a date, this argument should have a value of `3` in combination with the `Interval` argument with a value of `yyyy`. • To calculate a future date, use positive numbers. • To calculate a past date, use negative numbers.
`Original Date`	Date	Required	The original date to which you want to add the specified intervals

Returns

`DateAdd` returns the resulting date, which is the original date plus the defined interval(s). The return value is in the date data type.

Example

Suppose you are creating a bill that is due ten days after the current date. Your function might read as follows:

```
Value = DateAdd("d", 10, SYS_DateCurrent)
```

5.4.3 DateDiff

`DateDiff` identifies the span of time between two specified dates.

Syntax

```
DateDiff("Interval", Date1, Date2[, FirstDayOfWeek])
```

DateDiff arguments

Argument	Data type	Use	Description
`Interval`	String	Required	The string that represents the interval of time you want to use to measure the difference between the two dates Enter one of the following codes: • `yyyy`—Year • `q`—Quarter • `m`—Month • `y`—Day of year • `d`—Day • `w`—Weekday • `ww`—Week • `h`—Hour • `n`—Minute • `s`—Second You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`Date1`	Date	Required	The first date to use in calculation **Tip:** To create year-insensitive code, enclose `Date1` or `Date2` in double quotation marks and omit the year. The current year is inserted. If you want a specific year, specify it in `Date1` or `Date2`.

DateDiff arguments, continued

Argument	Data type	Use	Description
`Date2`	Date	Required	The second date to use in calculation **Tip:** To create year-insensitive code, enclose `Date1` or `Date2` in double quotation marks and omit the year. The current year is inserted. If you want a specific year, specify it in `Date1` or `Date2`.
`FirstDayOfWeek`	Integer	Optional	The code representing the day that is the first day of the week This argument affects calculations when the `Interval` argument is `w` (weekday) and `ww` (week). Enter one of the following codes: • 1—Sunday • 2—Monday • 3—Tuesday • 4—Wednesday • 5—Thursday • 6—Friday • 7—Saturday The default is `1`.

Returns

`DateDiff` returns the number of intervals specified that can fit between `Date1` and `Date2`.

- If `Interval` is weekday (`w`), `DateDiff` returns the number of weeks between the two dates. If `Date1` falls on a Monday, `DateDiff` counts the number of Mondays until `Date2`. It counts `Date2` but not `Date1`.
- If `Interval` is week (`ww`), `DateDiff` function returns the number of calendar weeks between the two dates. It counts the number of Sundays between `Date1` and `Date2`. It counts `Date2` if it falls on a Sunday, but it does not count `Date1`, even if it does fall on a Sunday.
- If `Date1` refers to a later point in time than `Date2`, the `DateDiff` function returns a negative number.

The return value is in the integer data type.

Example

Suppose that you want to calculate the number of days between the current date and the due date of a bill. Your function might read as follows:

```
Value = DateDiff("d", DueDate, SYS_DateCurrent)
```

5.4.4 DatePart

`DatePart` extracts a specific portion of information within a date (for example, the year).

Syntax

```
DatePart("Interval", Date[, FirstDayOfWeek, FirstWeekOfYear])
```

DatePart arguments

Argument	Data type	Use	Description
`Interval`	String	Required	A code representing the portion of the date you want to be returned Enter one of the following codes: • `yyyy`—Year • `q`—Quarter • `m`—Month • `y`—Day of year • `d`—Day • `w`—Weekday • `ww`—Week • `h`—Hour • `n`—Minute • `s`—Second You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`Date`	Date	Required	The date from which you want to extract the specified interval

DatePart arguments, continued

Argument	Data type	Use	Description
`FirstDayOfWeek`	Integer	Optional	The code representing the day that is the first day of the week This argument affects calculations when the `Interval` argument is `w` (weekday) and `ww` (week). Enter one of the following codes: • 1—Sunday • 2—Monday • 3—Tuesday • 4—Wednesday • 5—Thursday • 6—Friday • 7—Saturday The default is `1`.
`FirstWeekOfYear`	Integer	Optional	The code representing the week that is the first week of the year. Enter one of the following codes: • 1—The week of January 1 • 2—The first week in the new year that has at least four days in it • 3—The first full week in the new year The default is `1`.

Returns

`DatePart` returns the specified portion of the date. The return value is in the integer data type.

Example

Suppose you send a message to customers on their anniversary. You can use the `DatePart` function in the rule on the message.

In the Code Panel, you enter the following:

```
IF DatePart("m", DateAccountOpened) = DatePart("m", SYS_DateCurrent) THEN
INCLUDE
ENDIF
```

5.4.5 DateSerial

`DateSerial` assembles separate information about a year, month, and day to create a complete date.

Syntax

```
DateSerial(Year, Month, Day)
```

DateSerial arguments

Argument	Data type	Use	Description
`Year`	Integer	Required	The number that identifies the year Acceptable values range from `100` to `9999`.
`Month`	Integer	Required	The number that identifies the month Acceptable values range from `1` to `12`.
`Day`	Integer	Required	The number that identifies the day Acceptable values range from `1` to `31`.

> **Caution:** You receive an error message if the date specified by the three arguments falls outside the acceptable range of dates.

Returns

`DateSerial` returns the completed date. The return values are in the date data type.

5.4.6 DateValue

`DateValue` converts a date in the string data type to a date in the date data type.

> **Caution:** Packed, COBOL, ODBC timestamp, or time-only locale settings are not supported.

`DateValue` tests the date according to the following process, in up to three passes, until it can resolve the string:

1. `DateValue` attempts to resolve `DateString` using a literal type-for-type comparison with the locale setting (integer for integer, text for text, in the same order). After Pass 1, the engine processes only integer data. Text data returns a null value.

2. DateValue attempts to resolve DateString using an order comparison (for example, an integer month in the data can resolve even if the locale has the month in a text format).
3. DateValue attempts to resolve DateString using the default value of mm dd yy. If your string fails the third pass, then the engine returns a null value.

Syntax

```
DateValue("DateString")
```

DateValue argument

Argument	Data type	Use	Description
DateString	String	Required	A string expression representing a date and/or time Acceptable values range from January 1, 100 through December 31, 9999. The portion of the string that identifies the year must have two or four digits. You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.

Returns

DateValue returns a date.

By default, the format of the returned date is based on the **Date Format** setting on the locale object for the customer. However, if the function is used as a part of a date variable, and a different date setting is selected from the **Output format** list on the **Output Format** tab of the variable, the variable setting is used instead of the locale setting.

The return value is in the date data type.

5.4.7 Day

Day extracts the day information from a specified date.

Syntax

```
Day(Date)
```

Day argument

Argument	Data type	Use	Description
Date	Date	Required	The date from which you want to extract the day information

Returns

`Day` returns the day portion of a date as a value between `1` and `31` to represent the day of the month. If the `Date` argument contains a null string, the function returns a `0`. The return value is in the integer data type.

Example

If you are creating a list of birthdays for the month and you want to return only the specific day, you could enter the following code for a formula:

```
IF DatePart("m", EmployeeBirthdate) = DatePart("m", SYS_DateCurrent) THEN
Value = Day(EmployeeBirthdate)
ENDIF
```

5.4.8 IsDate

`IsDate` identifies whether a string is a date.

Syntax

```
IsDate("Expression")
```

IsDate argument

Argument	Data type	Use	Description
`Expression`	String	Required	The string you want to analyze to see whether it is a date. Acceptable values range from `January 1, 100` through `December 31, 9999`. However, this range can vary, based on your operating system. You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.

Returns

`IsDate` returns one of the following Boolean values:

- `TRUE`—The string expression can be converted to a date.
- `FALSE`—The string expression cannot be converted to a date.

5.4.9 Month

`Month` extracts the month information from a specified date.

Syntax

```
Month(Date)
```

Month argument

Argument	Data type	Use	Description
`Date`	Date	Required	The date from which you want to extract the month information

Returns

`Month` returns the month portion of a date as a value that ranges from `1` to `12` to represent the month of the year. If the `Date` argument contains a null string, the function returns a `0`. The return value is in the integer data type.

5.4.10 Weekday

`Weekday` identifies the day of the week on which a date occurs.

Syntax

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

Weekday arguments

Argument	Data type	Use	Description
`Date`	Date	Required	Any valid date expression you want to evaluate to determine a day of the week

Weekday arguments, continued

Argument	Data type	Use	Description
`FirstDayOfWeek`	Integer		A constant number that specifies the first day of the week Insert one of the following codes: • 1—Sunday • 2—Monday • 3—Tuesday • 4—Wednesday • 5—Thursday • 6—Friday • 7—Saturday The default is `1`.

Returns

`Weekday` returns one of the following values to identify the day of the week on which the date occurs:

- `0`—The date is a null string or the function was unsuccessful.
- 1—Sunday
- 2—Monday
- 3—Tuesday
- 4—Wednesday
- 5—Thursday
- 6—Friday
- 7—Saturday

The return value is in the integer data type.

5.4.11 Year

`Year` extracts the year information from a specified date.

Syntax

```
Year(Date)
```

Year argument

Argument	Data type	Use	Description
`Date`	Date	Required	The date from which you want to extract the year information

Returns

`Year` returns the year portion of a date as a value that ranges from `100` to `9999`, to represent the year extracted from the date. If the Date argument contains a null string, the function returns a `0`. The return value is in the integer data type.

5.5 General functions

General functions let you perform a variety of actions specific to the design and production environments. This section contains the references for all general functions.

- "ASCII" on the next page
- "Barcode" on page 114
- "CanImport" on page 115
- "Choose" on page 115
- "EBCDIC" on page 116
- "GetOS" on page 117
- "InJurisdiction" on page 118
- "IsEmpty" on page 119
- "IsNull" on page 119
- "IsNumeric" on page 120
- "LabelChar" on page 121
- "Message" on page 121
- "MessageCount" on page 123
- "ResetVariable" on page 124
- "RunMode" on page 124
- "SetColorCMYK" on page 125
- "SetColorRGB" on page 126
- "Shell" on page 127

- “StatusString” on page 128
- “Switch” on page 131

5.5.1 ASCII

`ASCII` converts a string from the EBCDIC to ASCII format. Conversions are based on the settings of the **ASCII To EBCDIC Translation** button settings in **System Settings**.

The `ASCII` built-in function works in the z/OS, OS/390, and Windows environments only.

For information on **System Settings**, see *System Administration* in the Exstream Design and Production documentation.

Syntax

```
Ascii(String[, ForceConversion])
```

ASCII arguments

Argument	Data type	Use	Description
`String`	String	Required	The string you want to convert You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`ForceConversion`	Boolean	Optional	A Boolean value that indicates whether you want to force the conversion if the data is in a format other than `Native` Enter one of the following Boolean values: • `TRUE`—Force the conversion. • `FALSE`—Do not force the conversion. The default is `FALSE`. **Caution:** Do not use the `ForceConversion` argument if the string is already in the ASCII format.

Returns

`ASCII` returns the ASCII version of the string entered for the `String` argument. The return value is in the string data type.

5.5.2 Barcode

`Barcode` converts a string into an encoded Interleaved 2 of 5 barcode.

For more information about Interleaved 2 of 5 barcodes, see *Creating Output* in the Exstream Design and Production documentation.

Syntax

```
Barcode("String", "MapString"[, BarcodeType])Barcode("String", "MapString"[,
BarcodeType])
```

Barcode arguments

Argument	Data type	Use	Description
`String`	String	Required	The alphanumeric string you want to make up the information in the barcode You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`MapString`	String	Required	The full map string for the barcode Map strings for Interleaved 2 of 5 barcodes must contain exactly 102 characters. • The first character of the `MapString` is used as the start character for the barcode. • The second character in the `MapString` is used as the end character for the barcode. **Note:** On z/OS and OS/390, the `MapString` is automatically converted from ASCII to EBCDIC. You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`BarcodeType`	Integer	Optional	A value that defines the barcode type For an Interleaved 2 of 5 barcode, enter `0`. The default is `0`.

Returns

`Barcode` returns an encoded Interleaved 2 of 5 barcode.

5.5.3 CanImport

`CanImport` tests to see whether a file can be loaded successfully with a specific placeholder variable.

For more information about placeholder variables, see *Using Data to Drive an Application* in the Exstream Design and Production documentation.

> **Caution:** `CanImport` operates accurately only from within the engine. If the `CanImport` function is used in Design Manager or LiveEditor, it always returns `FALSE`.

Syntax

```
CanImport(VariableName[ ,Index])
```

CanImport arguments

Argument	Data type	Use	Description
`VariableName`	Placeholder	Required	The name of the placeholder variable you want to test
`Index`	Integer	Optional	If the placeholder variable is an array, the number of the array element in the placeholder variable you want to receive the file The default is `1`.

Returns

`CanImport` returns one of the following Boolean values:

- `TRUE`—The file can be successfully loaded.
- `FALSE`—The file cannot be successfully loaded.

5.5.4 Choose

`Choose` selects a string value, based on customer information, from a list you create.

> **Tip:** Separately, the use of the `Choose` function and the use of a string lookup table can help you achieve similar results. Compare these two features to see which one is right for your needs.

For more information about using a string lookup table, see *Using Data to Drive an Application* in the Exstream Design and Production documentation.

Syntax

```
Choose(Index, "Value1", "Value2",...)Choose(Index, "Value1", "Value2",...)
```

Choose arguments

Argument	Data type	Use	Description
`Index`	Integer	Required	A variable that contains the numeric expression used to select a value from the remaining arguments The numeric value can range from `1` to `n` (where `n` is the number of available choices, based on the number of value arguments you include). If `Index` is not a whole number, it is rounded to the nearest whole number before it is evaluated.
`Value`	String	Required	The possible values that are selected based on the `Index` value You can repeat this argument as many times as needed to create a list of choices. You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.

Returns

`Choose` returns the value of the `Value` argument that corresponds to the value of the `Index` argument. If the index is less than `1` or greater than the number of `Value` arguments available, `Choose` returns a null string.

Example

Suppose your built-in function looks similar to the following:

```
Choose(Index, "Apples", "Oranges", "Bananas",)
```

The returns would be as follows:

- If the value of `Index` is `1`, `Choose` returns the string `Apples`.
- If the value of `Index` is `2`, `Choose` returns the string `Oranges`.
- If the value of `Index` is `5`, `Choose` returns a null string.

5.5.5 EBCDIC

`EBCDIC` converts a string from the ASCII to EBCDIC format. Conversions are based on the settings of the **ASCII To EBCDIC Translation** button settings in **System Settings**.

For information on **System Settings**, see *System Administration* in the Exstream Design and Production documentation.

Syntax

```
EBCDIC(String[, ForceConversion])
```

EBCDIC arguments

Argument	Data type	Use	Description
`String`	String	Required	The string you want to convert You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`ForceConversion`	Boolean	Optional	A Boolean value that indicates whether you want to force the conversion if the data is in a format other than `Native` Enter one of the following Boolean values: • `TRUE`—Force the conversion. • `FALSE`—Do not force the conversion. The default is `FALSE`. **Caution:** Do not use the `ForceConversion` argument if the string is already in the EBCDIC format.

Returns

`EBCDIC` returns the EBCDIC version of the string entered for the `String` argument. The return value is in the string data type.

5.5.6 GetOS

`GetOS` identifies the name of the operating system on which you are executing.

Tip: This function is often used as conditional logic for `Shell` functions.

For more information about `Shell` functions, see Shell.

Syntax

```
GetOS()
```

Caution: `GetOS` has no arguments. If you enter an argument, you receive an error message.

Returns

GetOS returns one of the following values:

- ZOS
- SOLARIS
- HPUX
- AIX
- WINDOWS
- LINUX
- ZLINUX

Return values are in the string data type.

5.5.7 InJurisdiction

InJurisdiction identifies whether the current customer is in a specified jurisdiction or group. This is especially useful if you need to handle jurisdictional information within rules or formulas.

Syntax

```
InJurisdiction("JurisdictionName"[, NameOrID])
```

InJurisdiction arguments

Argument	Data type	Use	Description
JurisdictionName	String	Required	The specified jurisdiction's name or ID You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
NameOrID	Integer	Optional	An integer used to specify whether JurisdictionName is a name or ID Enter one of the following values: • 0–JurisdictionNameis a name. • 1–JurisdictionNameis an ID. The default is 0.

Returns

`InJurisdiction` returns one of the following Boolean values:

- `TRUE`—The current customer is in the specified jurisdiction.
- `FALSE`—The current customer is not in the specified jurisdiction.

5.5.8 IsEmpty

`IsEmpty` identifies whether a variable value is zero or blank. This function cannot be used with array variables.

Syntax

```
IsEmpty(VariableName)
```

IsEmpty argument

Argument	Data type	Use	Description
`VariableName`	Any (non-array)	Required	The variable that contains the data you want to evaluate **Caution:** You can use only one variable with this function.

Returns

`IsEmpty` returns one of the following Boolean values:

- `TRUE`—The variable's value is zero or blank.
- `FALSE`—The variable's value is not zero or blank.

5.5.9 IsNull

`IsNull` identifies whether a variable value is a null string or zero.

> **Tip:** If you want to know whether a numeric variable is blank, rather than zero, map it as a string and make the numeric variable a formula: `VALUE = String`. You can then perform IsNull on the string.

Syntax

```
IsNull(VariableName)
```

IsNull argument

Argument	Data type	Use	Description
`VariableName`	Any (non-array)	Required	The variable containing the data you want to evaluate **Caution:** Only one variable can be used with this function.

Returns

`IsNull` returns one of the following Boolean values:

- `TRUE`—The variable is a null string or zero.
- `FALSE`—The variable is not a null string or zero.

5.5.10 IsNumeric

`IsNumeric` identifies whether a string can be used as a number. This function cannot be used to identify a date. If you need to identify a date, see the `IsDate` function.

Syntax

```
IsNumeric("Expression")
```

IsNumeric argument

Argument	Data type	Use	Description
`Expression`	String	Required	The string expression you want to evaluate You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.

Returns

`IsNumeric` returns one of the following Boolean values:

- `TRUE`—The expression is a number.
- `FALSE`—The expression is not a number.

5.5.11 LabelChar

`LabelChar` applies formatting to chart labels.

> **Tip:** You can also format chart labels on the **Legend/Label Properties** dialog box.

For more information about charts, see *Designing Customer Communications* in the Exstream Design and Production documentation.

Syntax

```
LabelChar("FormatType")
```

LabelChar argument

Argument	Data type	Use	Description
`FormatType`	String	Required	The format you want to apply to the chart labels Enter one of the following values: • `TAB`—Tab • `RIGHTTAB`—Tab-Right justified • `HANGINGTAB`—Hanging indent • `RETURN`—Begin a new line. You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.

Returns

`LabelChar` returns the custom formatting character associated with the format you apply. The return value is in the string data type.

5.5.12 Message

`Message` inserts a custom message into the engine message file.

> **Caution:** After the first use of the `Message` function, all parameters are fixed and you will not be able to change the text or other arguments for this message. If you need to make changes, you must create a separate message with a different message number.

Syntax

```
Message(Number, "Severity", "Text"[, MaxToShow, ExitRC, Error Level])
```

Message arguments

Argument	Data type	Use	Description
`Number`	Integer	Required	The number you want to assign to the message Acceptable values range from `1` to `10000`. **Note:** When you specify a message number, 10,000 is added to the number to adapt to the message numbering structure already in place.
`Severity`	String	Required	The severity level associated with the message Enter one of the following values: • `I`—Informational • `W`—Warning • `E`—Error • `S`—Severe You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`Text`	String	Optional	The text that makes up the message You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`MaxToShow`	Integer	Optional	The number of times the message should appear in the message file The default is blank (no limit).
`ExitRC`	Integer	Optional	The number that should be used as a return code. The return code is only used if the severity of the message is `S` (Severe). You must use a non-negative integer. The default is `12`.
`Error Level`	Integer	Optional	The value you want to apply to the 'SYS_CustInvalidDataLevel' system variable **Note:** If the `Error Level` value is greater than 10 and the output to the customer has not begun, the current customer is skipped. The default is `0`.

Returns

`Message` does not return a value.

5.5.13 MessageCount

`MessageCount` counts the number of campaign messages that have been qualified, sent, or excluded as the application runs.

Syntax

```
MessageCount("MessageType", CountType)
```

MessageCount arguments

Argument	Data type	Use	Description
`MessageType`	String	Required	The message type of the messages you want to count You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`CountType`	Integer	Required	The code that identifies the type of count you want to use Enter one of the following values: • 1—Count messages that qualified. • 2—Count messages what were sent. • 3—Count messages that were excluded. This means the messages were qualified, but were not sent. • 4—Count unique message that qualified. Multiple copies of the same message are counted only once. • 5—Count unique messages that were sent. Multiple copies of the same message are counted only once. • 6—Count unique messages that were excluded. Multiple copies of the same message are counted only once.

Returns

`MessageCount` returns the count of campaign messages, based on the message type and the count type you specified. If no messages of the specified `MessageType` are qualified, sent, or excluded, `-1` is returned.

The return value is in the integer data type.

5.5.14 ResetVariable

`ResetVariable` resets a variable to its default or initial value.

Syntax

```
ResetVariable(VariableName)
```

ResetVariable argument

Argument	Data type	Use	Description
`VariableName`	String	Required	The name of the variable you want to reset You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.

Returns

`ResetVariable` does not return a value.

5.5.15 RunMode

`RunMode` identifies the run mode of the engine.

Syntax

```
RunMode()
```

Caution: `RunMode` has no arguments. If you enter an argument, you receive an error.

Returns

`RunMode` returns one of the following values:

- `PRODUCTION`—The engine is in production mode.
- `LOCAL`—The engine is in local mode.

The return value is in the string data type.

5.5.16 SetColorCMYK

`SetColorCMYK` changes the CMYK color values of a named color based on customer data.

For the most reliable results using this function, make sure that `SetColorCMYK` is used at the start of an engine run. For example, suppose the variables used to control the color values are mapped to an initialization file. Exstream does not redraw objects that are perceived as static, and the colors of such objects are changed only by the first call to `SetColorCMYK`.

> **Note:** The `SetColorCMYK` function overrides spot colors and color tables.

Syntax

```
SetColorCMYK("ColorName", C-Value, M-Value, Y-Value, K-Value)
```

SetColorCMYK arguments

Argument	Data type	Use	Description
`ColorName`	String	Required	The name of the named color you are changing You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`C-Value`	Integer	Required	The Cyan value of the new color. Values must be between `0` and `100`.
`M-Value`	Integer	Required	The Magenta value of the new color. Values must be between `0` and `100`.
`Y-Value`	Integer	Required	The Yellow value of the new color. Values must be between `0` and `100`.
`K-Value`	Integer	Required	The Black value of the new color. Values must be between `0` and `100`.

Returns

`SetColorCMYK` returns one of the following Boolean values:

- `TRUE`—The named color you specified exists and the CMYK values are valid.
- `FALSE`—The color change request is invalid.

Example

The following formula specifies a color to change. If the color does not change, an error message appears in the engine message file.

```
IF NOT SetColorCMYK("Named Color", 100, 100, 100, 100) THEN
          MESSAGE (1, "E", "Unable to set color")
ENDIF
```

5.5.17 SetColorRGB

`SetColorRGB` changes the RGB color values of a named color based on customer data.

For the most reliable results using this function, make sure that `SetColorRGB` is used at the start of an engine run. For example, suppose the variables used to control the color values are mapped to an initialization file. Exstream does not redraw objects that are perceived as static, and the colors of such objects are only changed by the first call to `SetColorRGB`.

> **Note:** The `SetColorRGB` function overrides spot colors and color tables.

Syntax

SetColorRGB arguments

Argument	Data type	Use	Description
`ColorName`	String	Required	The name of the named color you are changing You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`R-Value`	Integer	Required	The Re value of the new color. Values must be between `0` and `255`.
`G-Value`	Integer	Required	The Green value of the new color. Values must be between `0` and `255`.
`B-Value`	Integer	Required	The Blue value of the new color. Values must be between `0` and `255`.

Returns

`SetColorRGB` returns one of the following Boolean values:

- `TRUE`—The named color you specified exists and the RGB values are valid.
- `FALSE`—The color change request is invalid.

Example

The following formula specifies a color to change. If the color does not change, an error message appears in the engine message file.

```
IF NOT SetColorRGB("Named Color", 100, 100, 100) THEN
          MESSAGE (1, "E", "Unable to set color")
ENDIF
```

5.5.18 Shell

`Shell` submits commands to the operating system command shell.

Caution: This function is not supported in mainframe systems.

Syntax

```
Shell("Command"[, SynchronousFlag])
```

Shell arguments

Argument	Data type	Use	Description
`Command`	String	Required	The full command string that specifies the execution you want the function to perform. The `Command` string cannot be longer than 1,024 characters. You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`SynchronousFlag`	Boolean	Optional	The Boolean value that indicates whether you want to hold the processing of the code until the command has completed execution. Enter one of the following Boolean conditions: • `TRUE`—The system waits for the execution to occur before the remaining code in the formula is processed. • `FALSE`—The function returns immediately, and the remainder of the formula is processed while the command is being executed. The default is `FALSE`.

Returns

`Shell` returns one of the following status codes:

Shell returns

Status code	Description
`0`	Success
`-1`	Error
`-2`	Command string exceeds the maximum (1,024 characters).
`-3`	File or path cannot be found (Windows only).
`-4`	The file specified is not an executable (Windows only).
`-5`	Not enough memory available (Windows only)

The return value is in the integer data type.

If Shell Returns a Value But Does Not Execute

In Windows, when the `Shell` function is executed, it is looking for a specific executable provided by an environment variable. If you are performing the `Shell` function and are returning a success value (`0`), but you are not receiving the correct output, check the `ComSpec` environment variable and make sure it points to either of the following:

```
(Drive letter):\WINNT\system32\cmd.exe (on Windows NT environment)
```

```
(Drive letter):\WINDOWS\system32\cmd.exe (on Windows 7 or later environment)
```

If you see a string in the **Variable Value** box that resembles `%SystemRoot%\system32\cmd.exe`, then delete the `%SystemRoot%` and hard code the path to it (for example, `C:\WINNT or C:\WINDOWS`).

In UNIX, the `Shell` function is searching for the command: `\usr\bin\sh -c`

5.5.19 StatusString

`StatusString` identifies the meaning of a status code by converting a status code into a status string.

Syntax

```
StatusString(StatusCode)
```

StatusString argument

Argument	Data type	Use	Description
`StatusCode`	Integer	Required	The integer representing a description of the I/O operation results

Returns

`StatusString` returns one of the following values:

StatusString status codes

StatusCode value	Returned string
`-2`	`No operation performed.`
`-1`	`User cancelled operation.`
`0`	`Operation successful.`
`1`	`Specified Exstream object not found.`
`2`	`Error writing file.`
`3`	`Error performing file record lookup.`
`4`	`Error performing file record update.`
`5`	`Error performing file record delete.`
`6`	`Exstream Data File is not of type ODBC.`
`7`	`No record layouts found for the Exstream Data File.`
`8`	`The operation attempted is not allowed for the record access mode.`
`9`	`Exstream Data File has no JDBC alias defined.`
`10`	`No privileges for the attempted operation.`
`11`	`Error reading an initialization Data File.`
`12`	`Error opening an Exstream Data File.`
`13`	`Error reading from an Exstream Data File.`
`14`	`Error closing an Exstream Data File.`
`15`	`Live document cannot be closed when the debugger is paused.`

StatusString status codes, continued

StatusCode value	Returned string
16	No license for this operation.
17	Data File of type ODBC is not allowed.
18	Key value unchanged, no lookup performed.
19	No key value, no lookup performed.
20	Data truncated writing output.
21	No start customer layout found.
22	The Exstream Data File type is incorrect for the requested operation.
23	The Exstream Data File is not columnar or delimited.
24	Data mapping error.
30	The file specified cannot be found.
31	The file specified is already open.
40	Mail not sent—cannot open MAPI32.DLL.
41	Mail not sent—bad DLL, cannot locate MAPISendMail.
42	Mail not sent—no recipient specified.
43	Mail not sent—MAPISendMail returns failure.
50	Live user not in Design Group.
51	Design Group is unknown.
52	Live authorization not setup.
66	Circular loop in Live actions detected—exiting current action.
67	Cancelled by Live system event function.
71	Unable to located a signature by ID or name.
72	Invalid certificate name.
73	Signature has been invalidated.
74	Signature has been erased.

StatusString status codes, continued

StatusCode value	Returned string
75	`The specified signature cannot be erased.`
99	`Internal error—please contact OpenText Support.`

The return value is in the string data type.

5.5.20 Switch

`Switch` allows you to evaluate a list of Boolean expressions and assign a specific value, based on the first Boolean expression in the list to be `TRUE`.

Syntax

```
Switch(Expr1, Value1[, Expr2, Value2,...])
```

Switch arguments

Argument	Data type	Use	Description
`Expr1`	Boolean	Required	The Boolean expression you want to evaluate
`Value1`	Any	Required	The value you want to return if the value of `Expr1` is `TRUE`
`Expr2`	Boolean	Optional	The Boolean expression you want to evaluate
`Value2`	Any	Optional	The value you want to return if the value of `Expr2` is `TRUE`

You can include as many expression and value pairs as needed.

Returns

`Switch` returns the value that corresponds to the first expression in the list that is `TRUE`. If none of the expressions are `TRUE`, the function returns a null string, a zero numeric value, or a `FALSE`. The return value is in the same data type as used for the `Value` argument.

5.6 I/O functions

I/O functions let you set input/output information for the engine. This section contains the references for all I/O functions.

- “CanImport” below
- “DDAFileMap” on the next page
- “DSNMap” on page 135
- “EvaluateXPath” on page 138
- “FileMap” on page 142
- “MapLayout” on page 144
- “Trigger” on page 146
- “WebServiceMap” on page 149

5.6.1 CanImport

`CanImport` tests to see whether a file can be loaded successfully with a specific placeholder variable.

For more information about placeholder variables, see *Using Data to Drive an Application* in the Exstream Design and Production documentation.

> **Caution:** `CanImport` operates accurately only from within the engine. If the `CanImport` function is used in Design Manager or LiveEditor, it always returns `FALSE`.

Syntax

```
CanImport(VariableName[ ,Index])
```

CanImport arguments

Argument	Data type	Use	Description
`VariableName`	Placeholder	Required	The name of the placeholder variable you want to test
`Index`	Integer	Optional	If the placeholder variable is an array, the number of the array element in the placeholder variable you want to receive the file The default is `1`.

Returns

CanImport returns one of the following Boolean values:

- TRUE—The file can be successfully loaded.
- FALSE—The file cannot be successfully loaded.

5.6.2 DDAFileMap

DDAFileMap changes file targets for data files to a DDA routine. If the specified data file is open, the function closes it and reopens it using the specified DDA routine. Optionally, you can perform a Trigger function after the file is opened.

This function overrides the settings in the DDAFILEMAP switch.

If you are using the DDAFileMap function in a Live application, after the DDAFileMap function is used, the function settings are used for the remainder of the LiveEditor session. For example, if the LiveDataFileOpen function is used after the DDAFileMap function, the file specified in the DDAFileMap function is used in place of the file specified in the LiveDataFileOpen settings.

Syntax

```
DDAFileMap(DataFileName, "Module", "Routine", BufSize [, "Param", Type,
Trigger])
```

DDAFileMap arguments

Argument	Data type	Use	Description
DataFileName	String	Required	The name of a data file. If the file has a rule that is executed during this timing, the rule is fired again. You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
Module	String	Required	The DLL/routine path name You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
Routine	String	Required	The function name in the DLL/routine You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
BufSize	Integer	Required	The number of bytes required for the output

DDAFileMap arguments, continued

Argument	Data type	Use	Description
`Param`	String	Optional	Any open parameters that apply You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`Type`	Integer	Required	The integer representing the type of programming used in the DDA routine Enter one of the following values: • `0`—DLL • `1`—COBOL • `2`—C • `3`—ASM • `5`—PL/I
`Trigger`	Boolean	Optional	The Boolean value that indicates whether you want run `Trigger` Enter one of the following Boolean conditions: • `TRUE`—Run the default mode for `Trigger`. The default action for `Trigger` depends on the type of data file you are using. • For customer driver or initialization files, the entire file is read. • For reference files, a lookup is performed. • For report files, write is performed. • `FALSE`—Do not run Trigger. The default is `FALSE`. **Note:** If you do not perform a `Trigger` after the file is opened, the DDAFILEMAP switch is overridden by the `DSNMap` and `FileMap` functions.

Returns

`DDAFileMap` returns one of the following status codes:

DDAFileMap status codes

Status code	Description
`0`	Success

DDAFileMap status codes, continued

Status code	Description
`1`	No such file or wrong file type
`2`	Error writing file
`3`	Error performing file record lookup
`4`	Error performing file record update
`5`	Error performing file record delete
`6`	The specified file is not ODBC.
`7`	No layouts are defined (no variables are mapped).
`8`	Function invalid for View, Join, or Distinct ODBC query
`9`	No JDBC Alias defined
`10`	No privileges for attempted operation
`30`	No such file or wrong file type
`32`	Web service cannot resolve host.
`33`	Web service cannot connect.
`34`	Web service SOAP fault
`35`	Web service other network error

The return value is in the integer data type.

5.6.3 DSNMap

`DSNMap` is available only if you have licensed the ODBC Access module. `DSNMap` specifies the DSN, passwords, and schema in the database at engine run time. Optionally, you can perform a `Trigger` function after the file is opened.

This function overrides the settings in the DSNMAP switch.

If you use this function as a part of a Live document, after the `DSNMap` function is used, the `DSNMAP` settings are used for the remainder of the LiveEditor session. For example, if the `LiveDataFileOpen` function is used, the `DSNMap` settings will be used instead of the `LiveDataFileOpen` settings.

Syntax

```
DSNMap("DataFileName", "DSN"[, "Username", "Password", "Schema", Trigger])
```

DSNMap arguments

Argument	Data type	Use	Description
`DataFileName`	String	Required	The name of the data file that uses the DSN. If the file has a rule that is executed during this timing, the rule is fired again. You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`DSN`	String	Required	The name of the new DSN You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`Username`	String	Optional	The user name for the database You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`Password`	String	Optional	The password for the database You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`Schema`	String	Optional	The schema of the database You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.

DSNMap arguments, continued

Argument	Data type	Use	Description
`Trigger`	Boolean	Optional	The Boolean value that indicates whether you want run `Trigger` Enter one of the following Boolean conditions: • `TRUE`—Run the default mode for `Trigger`. The default action for `Trigger` depends on the type of data file you are using. • For customer driver or initialization files, the entire file is read. • For reference files, a lookup is performed. • For report files, write is performed. • `FALSE`—Do not run `Trigger`. The default is `FALSE`. **Note:** If you do not perform a `Trigger` after the file is opened, the DSNMAP switch is overridden by the `DDAFileMap` and `FileMap` functions.

Returns

`DSNMap` returns one of the following status codes:

DSNMap status codes

Status code	Description
`0`	Success
`1`	No such file or wrong file type
`2`	Error writing file
`3`	Error performing file record lookup
`4`	Error performing file record update
`5`	Error performing file record delete
`6`	The specified file is not ODBC.
`7`	No layouts are defined (no variables are mapped).
`8`	Function invalid for View, Join, or Distinct ODBC query

DSNMap status codes, continued

Status code	Description
`9`	No JDBC Alias defined
`10`	No privileges for attempted operation
`30`	No such file or wrong file type
`32`	Web service cannot resolve host.
`33`	Web service cannot connect.
`34`	Web service SOAP fault
`35`	Web service other network error

The return value is in the integer data type.

5.6.4 EvaluateXPath

The `EvaluateXPath` function can be used only with schema model data files.

For more information about schema model data files, see *Using Data to Drive an Application* in the Exstream Design and Production documentation.

The `EvaluateXPath` function lets you evaluate the data within a schema model data file by using XPath expressions in order to gain broader access to the data that is within a schema model data file than you might be able to through typical mapping or filters.

For best results when using the EvaluateXPath function, you should be familiar with writing XPath expressions.

Syntax

```
EvaluateXpath(Array, "DataFileName", "DefaultNamespace", "XPathExpression",
[AdditionalNamespaces])
```

EvaluateXPath arguments

Argument	Data type	Use	Description
`Array`	Any	Required	The name of the array variable that you want to use to contain the returned data.

EvaluateXPath arguments, continued

Argument	Data type	Use	Description
`DataFileName`	String	Required	The name of the schema model data file that you want to evaluate. You can enter the value as a variable or as a string. You must enter string values in double quotation marks. Text that is entered without double quotation marks is interpreted as the name of a variable.
`DefaultNamespace`	String	Required	The namespace URI of the default namespace that is used in the schema model data file. For example: `www.hp.com/go/exstream` You can enter the value as a variable (either scalar or array) or as a string. You must enter string values in double quotation marks. Text that is entered without double quotation marks is interpreted as the name of a variable. If the data file content that you are evaluating does not belong to a namespace, you can use an empty string (`""`) to represent the namespace URI.
`XPathExpression`	String	Required	The XPath expression that you want to evaluate. As you define your XPath expression, keep in mind the following considerations: • All XPath expressions must be evaluated in the context of the root node. • If you want to limit the scope of the Xpath expression results to a specific customer, you must use a predicate to identify the customer node in the XPath expression. Otherwise, the function will evaluate the XPath expression against nodes from the entire data file. • If your data file uses multiple namespaces, you must specify the namespace prefix that is associated with each XPath location in the XPath expression. Otherwise, Exstream assumes that any location in the XPath expression that does not use a namespace prefix exists within the default namespace. • If there is a prefix in the XPath expression that does not belong to the namespace array, the function will fail. • Exstream does not validate the accuracy of your XPath code. • You can enter the value as a variable (either scalar or array) or as a string. You must enter string values in double quotation marks. Text that is entered without double quotation marks is interpreted as the name of a variable.

`AdditionalNamespaces`	String	Optional	An array variable that specifies any additional namespace URIs and namespace prefixes that are used by the schema model data file.
			Within the array variable, the syntax of the namespace URIs and prefixes must appear in the following format:
			`prefix:namespace URI`
			For example:
			`esx:www.hp.com/go/exstream`

Returns

The `EvaluateXPath` function returns a count of the elements that are returned in the array that you specified in the `Array` argument for the function. The return value is in the integer data type.

In the array variable, array elements that were pulled from the schema model data file that you specified in the function remain in their original data formats. Array values in the array variable start at an index of `1`.

Examples

There are many different ways that you can use the `EvaluateXPath` function. The following table includes different examples that illustrate the versatility of the `EvaluateXPath` function:

<table>
<tr><th>Example</th><th>Description</th></tr>
<tr><td>Data Filtering</td><td>

Suppose that the data source for your schema model data file is arranged as follows:

```
<BankStatements>
     <Customer Account="4172130">
          <Name>Thomas Reagan</Name>
          <Transactions>30</Transactions>
          <Bank1 name="Cust_B1">
               <Savings>
                    <Debit date="080114"
currency="USD">198.00</Debit>
                    <Credit date="080414"
currency="USD">398.48</Credit>
                    <Bank1>Cust_B2</Bank1>
               </Savings>
               <Checking>
                    <Debit date="080714"
currency="USD">107.57</Debit>
                    <Credit date="080814"
currency="USD">387.16</Credit>
                    <Credit date="080814"
currency="USD">278.90</Credit>
                    <Bank1>Check_B1</Bank1>
               </Checking>
          </Bank1>
     </Customer>
</BankStatements>
```

If you wanted to look up all the Bank1 credit transactions that were completed within the savings accounts for each customer in the schema model data file, the syntax for the `EvaluateXPath` built-in function might look similar to the following example:

```
EvaluateXPath(ResultsArrayVariable, “MyDataFile”, “”,
“/BankStatements/Customer/Bank1[“ & SYS_CustomerInRun &
“]/Savings/Credit”)
```

Notice that for this example, the `XPathExpression` argument uses the 'SYS_CustomerInRun' system variable in order to return results based on the customer that the engine is processing during the engine run when the function is encountered.

</td></tr>
</table>

Example	Description

Locate data outside the customer tags	Suppose that the data source for your schema model data file contains data that is outside of the scope of any defined customer tags, but you need to access that data for use in your application. The data source for your schema model data file might be arranged as follows:
	`<BankStatements xmlns="www.hp.com"`
	`xmlns:ex="www.hp.com/go/exstream">`
	`<ex:Init>`
	`<ex:RunID>12345</ex:RunID>`
	`</ex:Init>`
	`<ex:Customer Account="4172130">`
	`...`
	`</ex:Customer>`
	`</BankStatements>`
	If you needed to access the data in the `<ex:RunID>` tags, traditional mapping would not allow you to access that data because data mapping is customer-centric and the `<ex:RunID>` information exists outside of the customer tags. You will need to set up the `EvaluateXPath` function in order to access data that is outside of the customer data structure.
	Because the content also uses additional namespaces, you will also need to include a namespace array variable (`NamespaceArray`) that defines the following additional namespace and prefix from the data source:
	`ex:www.hp.com/go/exstream`
	The syntax for the `EvaluateXPath` built-in function might look similar to the following example:
	`EvaluateXPath(ResultsArray, “MyDataFile”, “www.hp.com”,`
	`“/BankStatements/ex:Init/ex:RunID”, NamespaceArray)`
	Notice that for this scenario, the function does not include the 'SYS_CustomerInRun' variable. Without the 'SYS_CustomerInRun' variable, the function returns the same result set regardless of when or where the function is included during the engine run.

Example	Description

Complete manual key/keyref lookups without having any key/keyref definitions in the data file	Suppose that the data source for your schema model data file is arranged as follows:
	`<root>`
	`     <Policy>`
	`          <Number>123456</Number>`
	`          <PolicyEffDate>Jan 1 2015</PolicyEffDate>`
	`     </Policy>`
	`     <Policy>`
	`          <Number>222222</Number>`
	`          <PolicyEffDate>Aug 1 2012</PolicyEffDate>`
	`     </Policy>`
	`     <Insured>`
	`          <ID>1</ID>`
	`          <PolicyNum>222222</PolicyNum>`
	`     </Insured>`
	`</root>`
	Also suppose that you want to look up the effective date (`<PolicyEffDate>`) of a customer policy based on the policy number of a specific insured customer (`<PolicyNum>`). Based on the order of the data, the effective date and the insured customer's policy number are kept in different locations in the XML file and do not have a keyref defined for Exstream to perform the lookup automatically.
	If you want to set up the `EvaluateXPath` function to complete a manual key/keyref lookup, the function syntax for this scenario might look similar to the following example:
	`EvaluateXPath(ResultsArray, “MyDataFile”, “”, “/root/Policy/PolicyEffDate[../Number = /root/Insured[“ & SYS_CustomerInRun & “]/PolicyNum]”)`
	Notice that for this example, the `XPathExpression` argument uses the 'SYS_CustomerInRun' system variable in order to return results based on the customer that the engine is processing during the engine run when the function is encountered.

5.6.5 FileMap

`FileMap` changes any file name specified in a data file's properties to a valid path and file name at run time. Optionally, you can perform a `Trigger` function after the file is opened.

The `FileMap` function overrides the settings in the FILEMAP switch.

If you use this function as a part of a Live document, after the `FileMap` function is used, the `FileMAP` settings are used for the remainder of the LiveEditor session. For example, if the `LiveDataFileOpen` function is used, the `FileMap` settings will be used instead of the `LiveDataFileOpen` settings.

> **Note:** You cannot use the `FileMap` function to open an ODBC data file. If you need to open an ODBC data file, you must use the `DSNMap` function.

For more information about the `DSNMap` function, see “DSNMap” on page 135.

Syntax

```
FileMap("DataFileName", "FileSystemFileName"[, Trigger])
```

FileMap arguments

Argument	Data type	Use	Description
`DataFileName`	String	Required	The name of a data file. If the file has a rule that is executed during this timing, the rule is fired again. You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`FileSystemFileName`	String	Required	The fully qualified name of the file You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`Trigger`		Optional	The Boolean value that indicates whether you want run `Trigger` Enter one of the following Boolean conditions: • `TRUE`—Run the default mode for `Trigger`. The default action for `Trigger` depends on the type of data file you are using. ◦ For customer driver or initialization files, the entire file is read. ◦ For reference files, a lookup is performed. ◦ For report files, write is performed. • `FALSE`—Do not run Trigger. The default is `FALSE`. **Note:** If you do not perform a `Trigger` after the file is opened, the FILEMAP switch is overridden by the `DDAFileMap` and `DSNMap` functions.

Returns

`FileMap` returns one of the following status codes:

FileMap status codes

Status code	Description
`0`	Success
`1`	Specified object not found

FileMap status codes, continued

Status code	Description
`2`	Error writing file
`3`	Error performing file record lookup
`4`	Error performing file record update
`5`	Error performing file record delete
`6`	The specified file is not ODBC.
`7`	No layouts are defined (no variables are mapped).
`8`	Function invalid for View, Join, or Distinct ODBC query
`9`	No JDBC Alias defined
`10`	No privileges for attempted operation
`17`	An ODBC data file cannot be used.
`30`	No such file or wrong file type
`32`	Web service cannot resolve host.
`33`	Web service cannot connect.
`34`	Web service SOAP fault
`35`	Web service other network error

The return value is in the integer data type.

5.6.6 MapLayout

`MapLayout` assigns variable values from a string based on a record layout. This function is useful if you need to decrease memory use when you are processing data files that contain many transaction records.

For example, suppose you have a customer driver file containing customers with 100,000 transaction records with 10 fields each. By default, in Design Manager, you would create 10 array variables with 100,000 data values each. This setup would cause you to have 1,000,000 data values in memory. By using `MapLayout`, you can have a single array variable with 100,000 data values, which are then split into 10 scalar variables for each transaction. This takes slightly more processing time, but significantly reduces the amount of memory required.

To use the `MapLayout` function:

1. Map an entire record in the customer driver or reference file to a buffer string array.
2. Call `MapLayout`, normally in a row rule, to map each string in the array to individual scalar variables defined in the auxiliary layout file.

> **Tip:** In the row rule, you can use the `Message` function to write a message for errors encountered when calling `MapLayout`.

For more information about using the `MapLayout` function with a data file, see *Using Data to Drive an Application* in the Exstream Design and Production documentation.

Syntax

```
MapLayout("Buffer", "DataFileName")
```

MapLayout arguments

Argument	Data type	Use	Description
`Buffer`	String	Required	The string variable containing all the values in the customer driver or reference file You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`Data File Name`	String	Required	The name of the auxiliary layout file containing the correct layout of the `Buffer` string. You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.

Returns

`MapLayout` returns one of the following status codes:

MapLayout status codes

Status code	Description
`0`	Success
`1`	No such data file
`22`	The data file type is incorrect for the requested operation.
`23`	The data file is not columnar or delimited.
`24`	Data mapping error

The return value is in the integer data type.

Example

In the following example, `Buffer_String` is the string array mapped to the entire customer driver file transaction record. "`Transaction Record`" is the name of the auxiliary layout data file that maps the individual transaction variables.

```
DIM iStatus AS INTEGER
iStatus = MAPLAYOUT(Buffer_String(SYS_TableRow), "Transaction Record")
IF iStatus <> 0 THEN
     MESSAGE(1, "E", "MapLayout fails, status " & iStatus)
ENDIF
INCLUDE
```

5.6.7 Trigger

`Trigger` allows you to specify your own timing for read and write operations during engine processing. For example, you can use a `Trigger` function to do the following:

- Read from a specified data file, such as a customer driver file, reference file, auxiliary layout file, initialization file, or postsort initialization file. These data files can point to data either in flat files (for example, .dat or .txt files) or in databases.
- Write to a specified report file. This can result in output generated to a flat file (for example, .dat or .txt files) or to an ODBC database.

This function can also be useful if you need to call a user function, update information for a table, create a report file, write to an ODBC database, or trigger other similar tasks.

Syntax

```
Trigger("DataFileName"[, Function, "SectionOrTagName"])
```

Caution: If a data file is renamed, references to the file name in a `Trigger` function in a formula are not updated. To make sure formulas are valid, update the changed data file name manually in the formula code where it is used.

Trigger arguments

Argument	Data type	Use	Description
`DataFileName`	String	Required	The name of a data file from which you want `Trigger` to read data You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`Function`	Integer	Optional	The operating mode `Trigger` uses. Use either the function name (such as `Read`) or a function number (such as `12`). The default is `0`. The `Function` argument supports the following I/O functions. You can specify either the function name, such as `Create` or `Read`, or the function number, such as `11` or `12`. **Tip:** To speed troubleshooting, use the function name instead of the function number. - `0/Mode 0`—Read (lookup) a reference file or write to a report file. If you use `Mode 0` with the XML input, you lose the parent/child relationships for tags. - `1/Mode 1`—Create a "section" in XML. Using this mode also requires you to specify a section or tag name in the `SectionOrTagName` argument. This argument is used only with the XML/JSON (Data) Output module. - `11/Create`—Create records in an ODBC table. This argument is used only with the ODBC Access module. - `12/Read`—Read an ODBC table. Mode 12 is the only function supported if the data file mapping uses distinct rows or contains a Join (a table View). This argument is used only with the ODBC Access module. - `13/Update`—Update existing records. Update fields should not be primary keys. This argument is used only with the ODBC Access module. - `14/Delete`—Delete records from an ODBC table. This argument is used only with the ODBC Access module. For more information about using the XML/JSON Input, the XML/JSON (Data) Output, or the ODBC Access modules, see *Using Data to Drive an Application* in the Exstream Design and Production documentation. The default is `0`.

Trigger arguments, continued

Argument	Data type	Use	Description
`SectionOrTagName`	String	Optional	The name of the section or tag that starts a new section in the data. You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable. This argument is used only when `Mode 1` is specified as `Function`. The default is `""` (null string).

Returns

`Trigger` returns one of the following status codes:

Trigger status codes

Status code	Description
`0`	Success
`1`	No such file or wrong file type
`2`	Error writing file
`3`	Error performing file record lookup
`4`	Error performing file record update
`5`	Error performing file record delete
`6`	The specified file is not ODBC.
`7`	No layouts are defined (no variables are mapped).
`8`	Function invalid for View, Join, or Distinct ODBC query
`9`	No JDBC Alias defined
`10`	No privileges for attempted operation
`30`	No such file or wrong file type
`32`	Web service cannot resolve host.
`33`	Web service cannot connect.
`34`	Web service SOAP fault
`35`	Web service other network error

The return value is in the integer data type.

5.6.8 WebServiceMap

`WebServiceMap` lets you change the mapping of the data source used for a Web service (SOAP) or Web service (RESTful) data file. For a SOAP Web service, the function allows you to modify the URL, SOAP action, and type of file transfer to use when sending file to a server. For a RESTful Web service, the function allows you to modify only the URL used. This function is not supported on the z/OS platform.

Syntax

```
WebServiceMap("DataFileName", "URL"[, "SOAP Action", Options, Trigger])
```

WebServiceMap arguments

Argument	Data type	Use	Description
`DataFileName`	String	Required	The name of the Exstream data file you want to remap You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`URL`	String	Required	The Web address of the server You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`SOAP Action`	String	Optional	The URI specifying the intent. If you enter nothing, Exstream assumes that the intent of the SOAP message is provided by the HTTP Request-URI. You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable. This argument is valid only for Web service (SOAP) data files. The default is a null value (`""`).

WebServiceMap arguments, continued

Argument	Data type	Use	Description
`Options`	Integer	Optional	The type of file transfer to use when sending file to a server. Enter one of the following values: • `0`—None • `1`—Secure transaction (https) • `2`—SOAP 1.1 specification • `4`—SOAP 1.2 specification The default is `0` (None). This argument is valid only for Web service (SOAP) data files.
`Trigger`	Boolean	Optional	The Boolean value that indicates whether you want run `Trigger` Enter one of the following Boolean conditions: • `TRUE`—Run the default mode for `Trigger`. The default action for `Trigger` depends on the type of data file you are using. • For reference files, a lookup is performed. • For report files, write is performed. • `FALSE`—Do not run `Trigger`. The default is `FALSE`. This argument is valid only for Web service (SOAP) data files.

Returns

`WebServiceMap` returns one of the following status codes:

WebServiceMap status codes

Status code	Description
`0`	Success
`12`	Error opening file
`30`	No such file or wrong file type

5.7 Math functions

Math functions let you perform mathematical operations, such as adding and averaging. This section contains the references for all math functions.

5.7.1 Abs

`Abs` calculates the absolute value of a number or equation. Absolute value is the unsigned value of a number. For example, the unsigned value of -1 is 1.

Syntax

```
Abs (Number)
```

Abs arguments

Argument	Data type	Use	Description
`Number`	Integer, Float, or Currency	Required	The value for which you want the absolute value If the variable you enter is an array, then you must specify the array element you want to use.

Returns

Abs returns the absolute value of the variable you specify. The returned value uses the same data type as the variable you enter for the Number argument.

Example

Suppose your application contains two variables which reference the customer's interest percentage rate: BeginPercentage, which shows the customer interest at the beginning of the year, and EndPercentage, which shows the customer interest at the end of the year. You want to show the customers how much their interest rate has changed within the past year by giving them the difference between the two values; however, you do not know which value is higher.

To find the absolute difference with the Abs function, you would enter the following:

```
Abs(EndPercentage - BeginPercentage)
```

5.7.2 Average

Average calculates the average value within a specified range of array elements.

5.7.3 Syntax

```
Average(Array[, Start, Stop])
```

Average arguments

Argument	Data type	Use	Description
Array	Integer, Float, or Currency	Required	The array variable containing the data you want to average
Start	Integer	Optional	The number of the first array element in the range you want to average The default is the first element in the array.
Stop	Integer	Optional	The number of the last array element in the range you want to average The default is the last element in the array.

Returns

Average returns the average value of the specified range of elements in the specified array. The return value is the same data type as the variable you specify in the Array argument.

Example

Suppose you want the average of an array named "Dependents", which contains the integers 5, 4, and 5. The syntax for `Average` might read as follows:

```
Average(Dependents[,1,3])
```

5.7.4 Contains

`Contains` locates the first element in an array that contains a specific value.

Syntax

```
Contains(Array, Target[, Start, Stop, Substring, NoCase, Sorted])
```

Contains arguments

Argument	Data type	Use	Description
`Array`	Any	Required	The array variable you want to search
`Target`	Same as `Array`	Required	The value for which you want to search
`Start`	Integer	Optional	The number of the first array element in the range you want to search The default is the first element in the array.
`Stop`	Integer	Optional	The number of the last array element in the range you want to search The default is the last element in the array.
`Substring`	Boolean	Optional	The Boolean value that indicates whether you want to also search substrings for the `Target` value Enter one of the following Boolean conditions: • `TRUE`—The target string can match only part of the value of the array element. • `FALSE`—The target string must exactly match the value of the array element. The default is `FALSE`.

Contains arguments, continued

Argument	Data type	Use	Description
`NoCase`	Boolean	Optional	The Boolean value that indicates whether you want to consider case when searching for the `Target` value Enter one of the following Boolean conditions: • `TRUE`—Strings are compared ignoring case. • `FALSE`—Strings must match exactly, including case. The default is `FALSE`.
`Sorted`	Boolean	Optional	The Boolean value that indicates whether the array is sorted. This determines whether the search is binary or linear. A binary search on data that is already sorted is faster than a linear search, particularly on arrays with ten or more elements. Enter one of the following Boolean conditions: • `TRUE`—The array content is sorted and you want to conduct a binary search on the array. If you specify `TRUE` and the array is not sorted, unexpected and inaccurate results may occur. • `FALSE`—The array content is not sorted and you want to conduct a linear search on the array. The default is `FALSE`.

Returns

`Contains` returns the index of the first array element that matches the value of the `Target` argument. If there is no match of the `Target` within the array, the function returns a zero (`0`). The return value is in the integer data type.

Example

Suppose you want to send a coupon for a free gift to any customer who has subscribed to your wireless program. Your built-in function might read as follows:

```
Contains(CustomerSubscriptions, Wireless, FALSE, TRUE, FALSE)
```

Since you know that any customer who has subscribed to your wireless program receives a non-zero value for this built-in function, you could use this function in combination with a rule to include the coupon if the value of `Contains` is greater than zero.

5.7.5 ContainsNot

`ContainsNot` locates the first element in an array that does not contain a specific value.

Syntax

```
ContainsNot(Array, Target[, Start, Stop, Substring, NoCase])
```

ContainsNot arguments

Argument	Data type	Use	Description
`Array`	Any	Required	The array variable you want to search
`Target`	Same as `Array`	Required	The value for which you want to search
`Start`	Integer	Optional	The number of the first array element in the range you want to search The default is the first element in the array.
`Stop`	Integer	Optional	The number of the last array element in the range you want to search The default is the last element in the array.
`Substring`	Boolean	Optional	The Boolean value that indicates whether you want to also search substrings for the `Target` value Enter one of the following Boolean conditions: • `TRUE`—The target string can match only part of the value of the array element. • `FALSE`—The target string must exactly match the value of the array element. The default is `FALSE`.
`NoCase`	Boolean	Optional	The Boolean value that indicates whether you want to consider case when searching for the `Target` value Enter one of the following Boolean conditions: • `TRUE`—Strings are compared ignoring case. • `FALSE`—Strings must match exactly, including case. The default is `FALSE`.

Returns

`ContainsNot` returns the index of the first element that does not match the value of the `Target` argument. If all the elements of the array match value of the `Target` argument, this function returns a `0` (zero) to indicate failure. The return value is in the integer data type.

5.7.6 Count

`Count` counts the number of elements in an array.

5.7.7 Syntax

```
Count(Array)
```

Count argument

Argument	Data type	Use	Description
`Array`	Any	Required	The array variable to count

Returns

`Count` returns the number of elements in the array you specified in the `Array` argument. The return value is in the integer data type.

Example

Suppose that you have an array that contains one element for each year that a customer has subscribed to your newsletter. If you want to find out how many years the customer has received your newsletter, you could use `Count` to count the number of elements in the array.

5.7.8 Hex

`Hex` converts an integer into a hexadecimal value.

Syntax

```
Hex(Number)
```

Hex argument

Argument	Data type	Use	Description
`Number`	Integer	Required	Any valid source expression **Note:** If `Number` is not a whole number, then it is rounded to the nearest number. The result is matched to the hexadecimal value.

Returns

`Hex` returns one of the following values:

- If the `Number` argument contains a number, then the function returns the hexadecimal value of the number.

- If the `Number` argument is null, then a null string is returned.
- If the `Number` argument is empty, then `0` is returned.

The return value is in the string data type.

5.7.9 Int

`Int` removes the fractional portion of any number. This function does not round the value.

Syntax

Int argument

Argument	Data type	Use	Description
`Number`	Float or Currency	Required	The number you want to convert to a whole number

Returns

`Int` returns the resulting whole number. The return value is in the integer data type.

5.7.10 Max

`Max` identifies the maximum value in a specified array.

Syntax

```
Max(Array[, Start, Stop])
```

Max arguments

Argument	Data type	Use	Description
`Array`	Any	Required	The array variable you want to search
`Start`	Integer	Optional	The number of the first array element in the range you want to search The default is the first element in the array.
`Stop`	Integer	Optional	The number of the last array element in the range you want to search The default is the last element in the array.

Returns

`Max` returns the maximum value in the array specified in the `Array` argument. The return value is in the same data type as the array specified in the `Array` argument.

5.7.11 Min

`Min` identifies the minimum value in a specified array.

Syntax

```
Min(Array[, Start, Stop])
```

Min arguments

Argument	Data type	Use	Description
`Array`	Any	Required	The array variable you want to search
`Start`	Integer	Optional	The number of the first array element in the range you want to search The default is the first element in the array.
`Stop`	Integer	Optional	The number of the last array element in the range you want to search The default is the last element in the array.

Returns

`Min` returns the minimum value in the array specified in the `Array` argument. The return value is in the same data type as the array specified in the `Array` argument.

5.7.12 Rnd

`Rnd` generates a random number between `0` and `1`. This function is helpful if you need to take random samples of customer data.

Syntax

```
Rnd[(Number)]
```

Rnd argument

Argument	Data type	Use	Description
`Number`	Numeric	Optional	The number you want to use to begin the random number generator To generate specific types of random numbers, do one of the following: • To return the next random number in the sequence, do not specify a value. • To return the most recently generated number, enter `0`. • To return the next random number in the sequence, enter a value that is greater than zero. • To return the same random number every time, enter a value that is less than zero. The default is `1.0`.

Returns

`Rnd` returns a value in the float data type.

5.7.13 Sum

`Sum` calculates the total of all the values in a specific range of elements within a numeric array.

Syntax

```
Sum(Array[, Start, Stop])
```

Sum arguments

Argument	Data type	Use	Description
`Array`	Integer, Float, or Currency	Required	The array variable containing the numeric data to total
`Start`	Integer	Optional	The number of the first array element in the range you want to add The default is the first element in the array.
`Stop`	Integer	Optional	The number of the last array element in the range you want to add The default is the last element in the array.

Returns

Sum returns the total of all the values in the specified range of elements within the array specified in the Array argument. The return value is in the same data type as the array specified in the Array argument.

5.7.14 WhenAverage

WhenAverage allows you to use a match array to identify specific elements, and then use the specified elements to identify and average the matching elements in a main array.

This function uses two arrays. The first array, Array, contains the values you want to use to calculate an average. The second array, MatchArray, is compared with a target. The function locates the elements in MatchArray that contain the Target value and then locates and averages the corresponding elements in Array.

Syntax

```
WhenAverage(Array, MatchArray, Target[, SubString, NoCase])
```

WhenAverage arguments

Argument	Data type	Use	Description
Array	Integer, Float, or Currency	Required	The array variable in which you want to search
MatchArray	Any	Required	The array variable within which you want to search for matches to the Target argument If the array specified for the MatchArray argument contains more elements than the array specified for the Array argument, only the number of elements in Array are searched in MatchArray. The remaining elements are ignored.
Target	Same as MatchArray	Required	The value for which you want to search
Substring	Boolean	Optional	The Boolean value that indicates whether you want to also search substrings for the **Target** value Enter one of the following Boolean conditions: • TRUE—The target string can match only part of the value of the array element. • FALSE—The target string must exactly match the value of the array element. The default is FALSE.

WhenAverage arguments, continued

Argument	Data type	Use	Description
`NoCase`	Boolean	Optional	The Boolean value that indicates whether you want to consider case when searching for the `Target` value - `TRUE`—Strings are compared ignoring case. - `FALSE`—Strings must match exactly, including case. The default is `FALSE`.

Returns

`WhenAverage` returns the average of the matching elements in the array specified for the `Array` argument. The return value is in the same data type as the array you specified for the `Array` argument.

Example

Suppose that the table below represents the two arrays.

Array	MatchArray
`10`	`!`
`20`	`@`
`30`	`!`
`40`	`@`
`50`	`!`

If you specify `Target` to be `!`, three matches are found in `MatchArray` and the corresponding elements in Array (`10`, `30`, and `50`) are averaged. This scenario would result in an average value of 30.

5.7.15 WhenCount

`WhenCount` counts a subset of items in an array triggered by a value of another array. It uses two arrays. The first array, `Array`, controls the number of elements searched in `MatchArray`. The second array, `MatchArray`, is searched for a match to `Target`.

Syntax

```
WhenCount(Array, MatchArray, Target[, SubString, NoCase])
```

WhenCount arguments

Argument	Data type	Use	Description
`Array`	Any	Required	The array variable containing the elements to be counted
`MatchArray`	Any	Required	The array variable within which you want to search for matches to the `Target` argument If the array specified for the `MatchArray` argument contains more elements than the array specified for the `Array` argument, only the number of elements in `Array` are searched in `MatchArray`. The remaining elements are ignored.
`Target`	Same as `MatchArray`	Required	The value for which you want to search
`Substring`	Boolean	Optional	The Boolean value that indicates whether you want to also search substrings for the `Target` value Enter one of the following Boolean conditions: • `TRUE`—The target string can match only part of the value of the array element. • `FALSE`—The target string must exactly match the value of the array element. The default is `FALSE`.
`NoCase`	Boolean	Optional	The Boolean value that indicates whether you want to consider case when searching for the `Target` value Enter one of the following Boolean conditions: • `TRUE`—Strings are compared ignoring case. • `FALSE`—Strings must match exactly, including case. The default is `FALSE`.

Returns

`WhenCount` returns the count of the number of elements that match `Target`. The return value is in the integer data type.

Example

Suppose that the table below represents the two arrays.

Array	MatchArray
`10`	`!`

Array	MatchArray
`20`	`@`
`30`	`!`
`40`	`@`
`50`	`!`

If you specify `Target` to be `!`, three matches are found and `3` is returned.

5.7.16 WhenSum

`WhenSum` allows you to use a match array to identify specific elements, and then use the specified elements to identify and add matching elements in a main array.

This function uses two arrays. The first array, `Array`, contains the values that are used to perform the computation. The second array, `MatchArray`, is searched for a match to `Target`. The elements that match the `Target` are stored and the data contained in the corresponding elements of `Array` are added together.

Syntax

```
WhenSum(Array, MatchArray, Target, SubString, NoCase)
```

WhenSum arguments

Argument	Data type	Use	Description
`Array`	Integer, Float, or Currency	Required	The array variable containing the elements to be added
`MatchArray`	Any	Required	The array variable within which you want to search for matches to the `Target` argument If the array specified for the `MatchArray` argument contains more elements than the array specified for the `Array` argument, only the number of elements in `Array` are searched in `MatchArray`. The remaining elements are ignored.
`Target`	Same as `MatchArray`	Required	The value for which you want to search

WhenSum arguments, continued

Argument	Data type	Use	Description
`Substring`	Boolean	Optional	The Boolean value that indicates whether you want to also search substrings for the `Target` value Enter one of the following Boolean conditions: • `TRUE`—The target string can match only part of the value of the array element. • `FALSE`—The target string must exactly match the value of the array element. The default is `FALSE`.
`NoCase`	Boolean	Optional	The Boolean value that indicates whether you want to consider case when searching for the `Target` value Enter one of the following Boolean conditions: • `TRUE`—Strings are compared ignoring case. • `FALSE`—Strings must match exactly, including case. The default is `FALSE`.

Returns

`WhenSum` returns the sum of the matching elements in the array specified for the `Array` argument. The return value is in the same data type as the array you specified for the `Array` argument.

Example

Suppose that the table below represents the two arrays.

Array	MatchArray
10	!
20	@
30	!
40	@
50	!

If you specify `Target` to be `!`, three matches are found the sum of the corresponding elements is returned:

```
10 + 30 + 50 = 90
```

5.8 String functions

String functions let you perform a variety of actions with string functions. This section contains the references for all string functions.

- “Asc” on the next page
- “ASCII” on the next page
- “Chr” on page 167
- “EBCDIC” on page 168
- “Format” on page 169
- “Instr” on page 169
- “Join” on page 170
- “LCase” on page 171
- “Left” on page 172
- “Len” on page 173
- “LTrim” on page 173
- “MapLayout” on page 174
- “Mid” on page 176
- “MidWord” on page 176
- “Replace” on page 178
- “Right” on page 178
- “RTrim” on page 179
- “Space” on page 180
- “Split” on page 181
- “StatusString” on page 182
- “Str” on page 184
- “StrConv” on page 185
- “String” on page 185
- “Trim” on page 186

- "UCase" on page 187
- "Val" on page 187

5.8.1 Asc

Asc identifies the character code of the first letter in a string.

> **Tip:** This function is similar to the Chr function; however, whereas the Asc function takes the character and identifies the character code, the Chr function takes the character code and identifies the character.

Syntax

```
Asc("String")Asc("String")
```

Asc argument

Argument	Data type	Use	Description
String	String	Required	The string value to be analyzed You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.

Returns

Asc returns the character code for the first letter in the string. The character code can have the following values:

- When working with SBCS applications, the character code can range from 0 to 255, based on the assigned values of an ASCII table.
- When working with DBCS applications, the character code can range from -32768 to 32767.
- If the string contains no characters, the function returns a zero (0).

The return value is in the integer data type.

5.8.2 ASCII

ASCII converts a string from the EBCDIC to ASCII format. Conversions are based on the settings of the **ASCII To EBCDIC Translation** button settings in **System Settings**.

The ASCII built-in function works in the z/OS, OS/390, and Windows environments only.

For information on **System Settings**, see *System Administration* in the Exstream Design and Production documentation.

Syntax

```
Ascii(String[, ForceConversion])
```

ASCII arguments

Argument	Data type	Use	Description
`String`	String	Required	The string you want to convert You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`ForceConversion`	Boolean	Optional	A Boolean value that indicates whether you want to force the conversion if the data is in a format other than `Native` Enter one of the following Boolean values: • `TRUE`—Force the conversion. • `FALSE`—Do not force the conversion. The default is `FALSE`. **Caution:** Do not use the `ForceConversion` argument if the string is already in the ASCII format.

Returns

`ASCII` returns the ASCII version of the string entered for the `String` argument. The return value is in the string data type.

5.8.3 Chr

`Chr` identifies the character associated with the specified character code.

> **Tip:** This function is similar to the `Asc` function; however, whereas the `Asc` function takes the character and identifies the character code, the `Chr` function takes the character code and identifies the character.

Syntax

```
Chr(Charcode)
```

Chr argument

Argument	Data type	Use	Description
`Charcode`	Integer	Required	The integer you want to analyze

Returns

`Chr` returns the character for the corresponding character code in the native character set. This means the characters returned can change for the same `Charcode` entered, depending on the platform you use.

The return values are in the string data type.

5.8.4 EBCDIC

`EBCDIC` converts a string from the ASCII to EBCDIC format. Conversions are based on the settings of the **ASCII To EBCDIC Translation** button settings in **System Settings**.

For information on **System Settings**, see *System Administration* in the Exstream Design and Production documentation.

Syntax

```
EBCDIC(String[, ForceConversion])
```

EBCDIC arguments

Argument	Data type	Use	Description
`String`	String	Required	The string you want to convert You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`ForceConversion`	Boolean	Optional	A Boolean value that indicates whether you want to force the conversion if the data is in a format other than `Native` Enter one of the following Boolean values: • `TRUE`—Force the conversion. • `FALSE`—Do not force the conversion. The default is `FALSE`. **Caution:** Do not use the `ForceConversion` argument if the string is already in the EBCDIC format.

Returns

`EBCDIC` returns the EBCDIC version of the string entered for the `String` argument. The return value is in the string data type.

5.8.5 Format

`Format` returns a string formatted according to instructions contained in a format expression. This function is similar to the `Str` function; however `Format` does not include a leading space for the sign of the returned number.

Syntax

```
Format(Expression[, "Format"])
```

Format arguments

Argument	Data type	Use	Description
`Expression`	Any	Required	Any valid source expression
`Format`	String	Optional	A valid named or user-defined format expression For more information about creating a custom format string, see *Using Data to Drive an Application* in the Exstream Design and Production documentation . You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable. The default is `""` (null string).

Returns

`Format` returns a string formatted according to instructions contained in a format expression. Any positive numbers that are reformatted into strings using the `Format` built-in function do not include a leading space reserved for the sign of the value.

5.8.6 Instr

`Instr` locates a string within a string.

Syntax

```
Instr([Start,] "String", "Target")
```

Instr arguments

Argument	Data type	Use	Description
`Start`	Integer	Optional	The number of the character position at which you want to begin the search The default is the first character position.
`String`	String	Required	The string within which you want to search You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`Target`	String	Required	The string for which you want to search You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.

Returns

`Instr` returns the starting byte position of the first occurrence of the specified target string within the specified string. The return values are in the integer data type. If the `Target` is not found, or if the `Start` position is greater than the `Target`, the function returns a `0`. If the `Target` has a length of zero, the `Start` position is returned.

5.8.7 Join

`Join` concatenates all the elements of a string array together into a single string, optionally separating each element with a delimiter.

Syntax

```
Join(Array[, "Delimiter"])
```

Join arguments

Argument	Data type	Use	Description
`Array`	String	Required	The array variable containing the elements you want to concatenate You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`Delimiter`	String	Optional	The character you want to use as a delimiter **Tip:** If you do not want spaces between the array elements, use double quotation marks as the delimiter. You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable. The default is a space.

Returns

`Join` returns the concatenated string. The return value is in the string data type.

5.8.8 LCase

`LCase` converts all uppercase characters in a string to lowercase characters. Any lowercase characters and numeric characters remain unchanged.

Syntax

```
LCase("String")
```

LCase argument

Argument	Data type	Use	Description
`String`	String	Required	The string that contains the characters you want to return as lowercase You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.

Returns

LCase returns one of the following values:

- If the function was successful, the function returns a string that has been converted to lowercase characters.
- If String contains a null value, the function returns a null string.

The return value is in the string data type.

5.8.9 Left

Left extracts a substring, based on a specific number of characters, from the left of a string.

Syntax

```
Left("String", Count)
```

Left arguments

Argument	Data type	Use	Description
String	String	Required	The string expression from which you want to extract a substring You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
Count	Integer	Required	The number of characters to extract The count begins from the character furthest to the left and continues to the right. **Tip:** If you need to determine the number of characters within a string, then use the Len function.

Returns

Left returns one of the following values:

- If Count is a non-zero value, the function returns the extracted substring, containing the specified number of characters.
- If Count is 0, the function returns a zero-length string.
- If Count is greater than the number of characters in the string, the entire string is returned.

The return value is in the string data type.

5.8.10 Len

`Len` identifies the length of a string by counting the number of characters within a string.

Syntax

```
Len("String")
```

Len argument

Argument	Data type	Use	Description
`String`	String	Required	The string expression you want to analyze You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.

Returns

`Len` returns one of the following values:

- If the function is successful, the function returns a count that represents the length of `String`.
- If `String` contains a null value, the function returns a `0`.

The return value is in the integer data type.

5.8.11 LTrim

`LTrim` trims the leading spaces or blanks from a string.

> **Note:** You can also specify that blanks are to be trimmed in the **Format** list on the **Data Area Properties** dialog box during data mapping. This is a more efficient method than using a formula, function, or rule.

For more information about mapping data, see *Using Data to Drive an Application* in the Exstream Design and Production documentation.

Syntax

```
LTrim("String")
```

LTrim argument

Argument	Data type	Use	Description
`String`	String	Required	The string expression from which to trim the leading spaces or blanks You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.

Returns

`LTrim` returns one of the following values:

- If the function is successful, the function returns the trimmed string.
- If `String` contains a null value, the function returns a null string.

The return value is in the string data type.

5.8.12 MapLayout

`MapLayout` assigns variable values from a string based on a record layout. This function is useful if you need to decrease memory use when you are processing data files that contain many transaction records.

For example, suppose you have a customer driver file containing customers with 100,000 transaction records with 10 fields each. By default, in Design Manager, you would create 10 array variables with 100,000 data values each. This setup would cause you to have 1,000,000 data values in memory. By using `MapLayout`, you can have a single array variable with 100,000 data values, which are then split into 10 scalar variables for each transaction. This takes slightly more processing time, but significantly reduces the amount of memory required.

To use the `MapLayout` function:

1. Map an entire record in the customer driver or reference file to a buffer string array.
2. Call `MapLayout`, normally in a row rule, to map each string in the array to individual scalar variables defined in the auxiliary layout file.

> **Tip:** In the row rule, you can use the `Message` function to write a message for errors encountered when calling `MapLayout`.

For more information about using the `MapLayout` function with a data file, see *Using Data to Drive an Application* in the Exstream Design and Production documentation.

Syntax

```
MapLayout("Buffer", "DataFileName")
```

MapLayout arguments

Argument	Data type	Use	Description
`Buffer`	String	Required	The string variable containing all the values in the customer driver or reference file You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`Data File Name`	String	Required	The name of the auxiliary layout file containing the correct layout of the `Buffer` string. You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.

Returns

`MapLayout` returns one of the following status codes:

MapLayout status codes

Status code	Description
`0`	Success
`1`	No such data file
`22`	The data file type is incorrect for the requested operation.
`23`	The data file is not columnar or delimited.
`24`	Data mapping error

The return value is in the integer data type.

Example

In the following example, `Buffer_String` is the string array mapped to the entire customer driver file transaction record. `"Transaction Record"` is the name of the auxiliary layout data file that maps the individual transaction variables.

```
DIM iStatus AS INTEGER
iStatus = MAPLAYOUT(Buffer_String(SYS_TableRow), "Transaction Record")
IF iStatus <> 0 THEN
     MESSAGE(1, "E", "MapLayout fails, status " & iStatus)
ENDIF
INCLUDE
```

5.8.13 Mid

`Mid` extracts a substring, based on a specific number of characters, from the middle of a string.

Syntax

```
Mid("String", Start[, Count])
```

Mid arguments

Argument	Data type	Use	Description
`String`	String	Required	The string expression from which you want to extract a substring You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`Start`	Integer	Required	The number of the character position at which the substring begins
`Count`	Integer	Optional	The number of characters to extract The count begins from the position you enter for the `Start` argument and continues to the right. By default, the substring returned begins at the `Start` position and completes the string. **Tip:** If you need to determine the number of characters within a string, then use the `Len` function.

Returns

`Mid` returns one of the following values:

- If the function is successful, the function returns the extracted substring, containing the specified number of characters.
- If `String` contains a null value, the function returns a null string.
- If the `Start` position is greater than the number of characters in string, the function returns a null string.

  The return value is in the string data type.

5.8.14 MidWord

`MidWord` extracts a substring, based on a specific number of words, from the middle of a string.

Syntax

```
MidWord("String", Start[, Count])
```

MidWord arguments

Argument	Data type	Use	Description
`String`	String	Required	The string expression from which you want to extract the substring You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`Start`	Integer	Required	The number of the first word position at which the substring begins
`Count`	Integer	Optional	The number of words to return in the substring The count begins from the position you enter for the `Start` argument and continues to the right. By default, all the words from the `Start` position to the end of the string are returned.

Returns

`MidWord` returns one of the following values:

- If the function is successful, the function returns the extracted substring, containing the specified number of words.
- If `String` contains a null value, the function returns a null string.
- If the `Start` position is greater than the number of characters in string, the function returns a null string.

  The return value is in the string data type.

Example

Suppose that a customer's full name is mapped to a single variable, such as -CustomerName.- If you need only the first name, you can use the `MidWord` function to extract the customer's first name from the variable. The syntax to use the `MidWord` function for this scenario might read as follows:

```
Value = MidWord(CustomerName, 1,1)
```

5.8.15 Replace

`Replace` replaces instances of a substring with another substring. This function can be helpful with name or address changes.

Syntax

```
Replace("String", "Find", "Replace"[, Start, Count])
```

Replace arguments

Argument	Data type	Use	Description
`String`	String	Required	The string you want to search You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`Find`	String	Required	The substring you want to replace You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`Replace`	String	Required	The substring you want to apply, which replaces the `Find` substring You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`Start`	Integer	Optional	The number of the first character position at which you want to begin the search The default is the first character position.
`Count`	Integer	Optional	The number of substitutions to perform in the string. If you do not want to limit the number of substitutions, enter `-1`. The default is `-1`.

Returns

`Replace` returns the original string, along with the found and replaced substrings.

5.8.16 Right

`Right` extracts a substring, based on a specific number of characters, from the right of a string.

Syntax

```
Right("String", Count)
```

Right arguments

Argument	Data type	Use	Description
`String`	String	Required	The string expression from which you want to extract a substring You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`Count`	Integer	Required	The number of characters you want to extract The count begins from the character furthest to the right and continues to the left. **Tip:** If you need to determine the number of characters within a string, then use the `Len` function.

Returns

`Right` returns the extracted substring, containing the specified number of characters. The return value is in the string data type.

- If `Count` is greater than or equal to the number of characters in the string, the function returns the entire string.
- If `Count` is equal to zero, the function returns a null string.
- If `String` contains a null value, the function returns a null string.

The return value is in the string data type.

5.8.17 RTrim

`RTrim` trims the trailing spaces or blanks from a string.

> **Note:** You can also specify that blanks are to be trimmed in the **Format** list on the **Data Area Properties** dialog box during data mapping. This is a more efficient method than using a formula, function, or rule.

For more information about formatting data area properties, see *Using Data to Drive an Application* in the Exstream Design and Production documentation.

Syntax

```
RTrim("String")
```

RTrim argument

Argument	Data type	Use	Description
`String`	String	Required	The string expression from which to trim the trailing spaces or blanks You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.

Returns

`RTrim` returns one of the following values:

- If the function is successful, the function returns the trimmed string.
- If `String` contains a null value, the function returns a null string.

The return value is in the string data type.

5.8.18 Space

`Space` returns a string containing the specified number of spaces. This function is useful for formatting output and clearing data in fixed-length strings. It can also be used to insert spaces between data items, such as category headings.

Syntax

```
Space(Count)
```

arguments

Argument	Data type	Use	Description
`Count`	Integer	Required	The number of spaces in the string

Returns

`Space` returns a string containing the specified number of spaces.

5.8.19 Split

`Split` divides a delimited string into an array of substrings.

Tip: If you have a string that is not delimited and you want to parse each character of the string into a separate array element, you should use a `FOR loop` instead of the `Split` function.

Syntax

```
Split(Array, "String"[, "Delimiter", Limit])
```

Split arguments

Argument	Data type	Use	Description
`Array`	String	Required	The array variable in which you want to place the resulting substrings You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`String`	String	Required	The source string you want to split into substrings **Note:** If the maximum number of array elements to create is exceeded, the remainder of the string is placed in the final element. You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`Delimiter`	String	Optional	The character you want to use as a delimiter You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable. The default is a space. If you enter a null value (`""`) or an invalid delimiter, the entire string is moved into the first array element.
`Limit`	Integer	Optional	The maximum number of elements that can be created. By default, there is no limit.

Returns

`Split` returns the number of substrings created. The return value is in the integer data type.

5.8.20 StatusString

`StatusString` identifies the meaning of a status code by converting a status code into a status string.

Syntax

```
StatusString(StatusCode)
```

StatusString argument

Argument	Data type	Use	Description
`StatusCode`	Integer	Required	The integer representing a description of the I/O operation results

Returns

`StatusString` returns one of the following values:

StatusString status codes

StatusCode value	Returned string
-2	`No operation performed.`
-1	`User cancelled operation.`
0	`Operation successful.`
1	`Specified Exstream object not found.`
2	`Error writing file.`
3	`Error performing file record lookup.`
4	`Error performing file record update.`
5	`Error performing file record delete.`
6	`Exstream Data File is not of type ODBC.`
7	`No record layouts found for the Exstream Data File.`
8	`The operation attempted is not allowed for the record access mode.`
9	`Exstream Data File has no JDBC alias defined.`

StatusString status codes, continued

StatusCode value	Returned string
10	No privileges for the attempted operation.
11	Error reading an initialization Data File.
12	Error opening an Exstream Data File.
13	Error reading from an Exstream Data File.
14	Error closing an Exstream Data File.
15	Live document cannot be closed when the debugger is paused.
16	No license for this operation.
17	Data File of type ODBC is not allowed.
18	Key value unchanged, no lookup performed.
19	No key value, no lookup performed.
20	Data truncated writing output.
21	No start customer layout found.
22	The Exstream Data File type is incorrect for the requested operation.
23	The Exstream Data File is not columnar or delimited.
24	Data mapping error.
30	The file specified cannot be found.
31	The file specified is already open.
40	Mail not sent—cannot open MAPI32.DLL.
41	Mail not sent—bad DLL, cannot locate MAPISendMail.
42	Mail not sent—no recipient specified.
43	Mail not sent—MAPISendMail returns failure.
50	Live user not in Design Group.
51	Design Group is unknown.
52	Live authorization not setup.

StatusString status codes, continued

StatusCode value	Returned string
66	`Circular loop in Live actions detected—exiting current action.`
67	`Cancelled by Live system event function.`
71	`Unable to located a signature by ID or name.`
72	`Invalid certificate name.`
73	`Signature has been invalidated.`
74	`Signature has been erased.`
75	`The specified signature cannot be erased.`
99	`Internal error—please contact OpenText Support.`

The return value is in the string data type.

5.8.21 Str

`Str` converts an integer into a string. When numbers are converted to strings, a leading space is reserved for the sign. If a number is positive, then the plus sign (+) is implied, so the sign is not included.

> **Tip:** The `Format` function is similar the `Str` function, but `Format` does not include a leading space for the sign. Use `Format` to convert values such as dates, times, and currency.

Syntax

```
Str(Number)
```

Str argument

Argument	Data type	Use	Description
`Number`	Integer	Required	Any valid numeric expression

Returns

`Str` returns a string representation of a number. The return value is in the string data type.

5.8.22 StrConv

`StrConv` converts a string to use a specific case (uppercase, lowercase, init-cap all words, or init-cap with exceptions) based on the argument specifications.

Syntax

```
StrConv("String", Conversion)
```

StrConv arguments

Argument	Data type	Use	Description
`String`	String	Required	The string expression you want to convert You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.
`Conversion`	Integer	Required	An integer that represents the type of conversion you want to apply to the string Enter one of the following values: • 1—Convert the string to uppercase characters. • 2—Convert the string to lowercase characters. • 3—Convert the first letter of each word to uppercase. • 4—Convert the string to initial caps.

Returns

`StrConv` returns the converted string. The return value is in the string data type.

5.8.23 String

`String` returns a string containing the first character of the source string, repeated a specified number of times.

Syntax

```
String(Count, "String")
```

String arguments

Argument	Data type	Use	Description
`Count`	Integer	Required	The length of the returned string. This is how many times the character repeats. **Note:** If `Count` is equal to zero, the function returns a null string.
`String`	String	Required	The string expression. The first character of this expression is used to create the output string. You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.

Returns

`String` returns a string containing the first character of the source string, repeated a specified number of times. The return value is in the string data format.

5.8.24 Trim

`Trim` trims the leading and trailing spaces or blanks from a string.

> **Note:** You can also specify that blanks are to be trimmed in the **Format** list on the **Data Area Properties** dialog box during data mapping. This is a more efficient method than using a formula, function, or rule.

For more information about formatting data area properties, see *Using Data to Drive an Application* in the Exstream Design and Production documentation.

Syntax

```
Trim("String")
```

Trim argument

Argument	Data type	Use	Description
`String`	String	Required	The string expression from which to trim the leading and trailing spaces or blanks You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.

Returns

`Trim` returns one of the following values:

- If the function is successful, the function returns the trimmed string.
- If `String` contains a null value, the function returns a null string.

The return value is in the string data type.

5.8.25 UCase

`UCase` converts all lowercase characters in a string to uppercase characters. Any uppercase characters and numeric characters remain unchanged.

Syntax

```
UCase("String")
```

UCase argument

Argument	Data type	Use	Description
`String`	String	Required	The string expression being converted to uppercase You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.

Returns

`UCase` returns one of the following values:

- If the function was successful, the function returns a string that has been converted to uppercase characters.
- If `String` contains a null value, the function returns a null string.

The return value is in the string data type.

5.8.26 Val

`Val` returns a floating number from a string argument. The `Val` function stops reading the string at the first character which the function cannot recognize as numeric. Symbols and characters which are often considered parts of numeric values, such as dollar signs and commas, are not recognized by the `Val` function. However, spaces are recognized and do not hinder the function.

Syntax

```
Val("String")
```

Val argument

Argument	Data type	Use	Description
`String`	String	Required	Any valid string expression to return a numeric value You must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.

Returns

`Val` returns one of the following values:

- If the function was successful, the function returns a floating number from the specified string argument.
- If `String` does not contain a number, `Val` returns a `0`.
- If `String` contains a null value, the function returns a null string.

The return value is in the float data type.

Chapter 6: Creating Effective and Efficient Code in Exstream

When you create rules, formulas, and functions, Exstream allows you to use generally accepted programming practices, modeled after Visual Basic, to create the code for logic. You can enter logic in Exstream using either of the following methods:

- **Entering code directly into Exstream**—When you enter code for logic in Exstream, keep in mind that although code in Exstream is modeled after Visual Basic, there are some key differences, especially in the data types and code you can use.
- **Using coding short-cut buttons to enter code**—The V, >=, Fx, and fx buttons below the coding areas in Exstream let you enter variables, conditions, built-in functions and Library functions exactly as they appear in your database. An added benefit of using these short-cut buttons is that it makes sure the names of variables and functions are spelled correctly.

For more information about building code, see the chapter for the object you are creating.

This chapter is meant to be a reference to help you create effective and efficient code.

6.1 About Exstream Code

This section outlines the key differences between Exstream code and Visual Basic.

6.1.1 Data Type Differences

While the data types used in Exstream are comparable to those used in Visual Basic code, there are differences in the terminology used to define each data type. The following table outlines the Visual Basic code data type and the Exstream equivalent:

Visual Basic data types and their Exstream equivalents

Data type	Visual Basic Code	Exstream code
Alpha	`string, char`	`string, Placeholder, TaggedText, FormattedText`
Numeric	`Integer, long, short, byte, sbyte, ushort, uinteger, ulong`	`Integer`
Alphanumeric	`double`	`Float`

Visual Basic data types and their Exstream equivalents, continued

Data type	Visual Basic Code	Exstream code
Currency or Decimal	`decimal`	`Currency`
Date	`date`	`date`
Boolean	`Boolean`	`Boolean`

6.1.2 Coding Differences

Some Visual Basic coding abilities function differently when used in Exstream. The following table outlines the Visual Basic ability and the Exstream equivalent: You must enter string values in double quotation marks.

Key differences between Visual Basic code and logic code used in Exstream

Code use	Visual Basic code	Exstream code
Comments	' (single quotation mark) Makes the remaining information in the same line of code a comment	// Makes the remaining information in the same line of code a comment /**/ Makes a inside comment. Mark the beginning of the comment with /* and and the comment with */
Multi-line statements	_ (underscore) Multi-line statements require a continuation character.	Exstream does not require a continuation character for multi-line statements.
Multiple statements per line	: (colon) Multiple statements that appear on a single line must be separated with a : (colon).	Exstream does not support multiple statements per line.
String concatenation	& (ampersand) and + (plus sign)	& (ampersand)
Enum, object, or property support	Available	Exstream does not support enum, object, or property support.
Variable names	Variables can be numeric strings.	Exstream requires at least one non-numeric character in any variable name.
Integer operators	+ - * / mod and bit operators	Exstream does not support bit operators.

Keep in mind that you must enter string values in double quotation marks. Text entered without double quotation marks will be interpreted as the name of a variable.

6.1.3 "Is Like" Conditions

If you selected **is like** as a condition, you can also use any combination of the following symbols with **is like** to make comparisons faster and more precise.

is like conditions

Symbol	Result	Example
[]	Encloses individual characters or ranges to match	Example of a range match: `"Example" Like "[A-FG-Z]"` looks for matches with characters A through F or G through Z. Example of a character match: `"Example" Like "[AB?]"` lets you look for a match with A, B, or ?
!	Looks for a match with any characters except the ones you specify	`"D" Like "[!A-C]"` returns true.
#	Looks for a match with a single number (0-9)	`"A5B" Like "A#B"` returns true.
*	Looks for a match with a substring	`"AB*" Like "ABCDEF"` returns true.
?	Looks for a match with any single character	`"ABC" Like "A?B"` returns true.
-	Looks for a match with a range of characters	`"[A-C]" Like "ABC"` returns true.

6.2 Creating Efficient and Readable Code

Inefficient code slows engine processing and makes the code difficult to read, especially when you are debugging and troubleshooting. The following sections outline some tips to help you keep your code efficient and readable.

To create more efficient code, follow these guidelines:

- Use the fewest number of statements possible. For example, if you find that it takes a substantial number of statements to exclude an object using a rule, consider whether it would require less code to create the rule to include that object.
- Avoid unnecessary repetition of blocks of code.

To make your code more readable, follow these guidelines:

- Use indentations to indicate nesting and code that is integral to specific statements.
- Use parentheses to separate specific pieces of code.

For example:

```
curTotal=(curTotal-intResults)
```

- Use spaces between mathematical and concatenation operators and the objects they operate upon.
- Use comments to separate blocks of functionally unique code.

6.2.1 Using Variables Effectively in Code

Almost every rule, formula, or function you create will use variable information. To make sure you get the best results when using variables, follow these guidelines:

- Avoid declaring unnecessary variables.
- Declare all local variables at the top of your code.
- Give variables meaningful names. It can be helpful to describe the variable's use or return value either as a part of the name or as a part of the comments.
- Variable formulas can be a function of themselves. Using a variable as a function of itself is also useful because it supports accumulators without using counters.

  For example:

  ```
  VariableA=VariableA + 1
  ```

> **Caution:** Exstream does not support exceptions for overflows or underflows. Use caution when creating formulas that can result in overflow. If the formula results in values of more than 2 billion, you should not use an integer variable.

For more information about setting the type of data that will be returned by a variable, see *Using Data to Drive an Application* in the Exstream Design and Production documentation.

6.2.2 Avoiding Circular References When Creating Library Function Logic

When you package an application, the packaging process checks for circular references. If a circular reference is found, you receive a severe error. You must correct the logic and repackage to obtain a working package file.

Circular references can occur between functions and other functions, formula variables and other formula variables, or functions and formula variables.

Appendix A: Built-in Function Quick Reference

This appendix gives a quick reference for using and defining built-in functions in your code. Built-in function references in this appendix include the syntax for each built-in function and indicate, with a subscript, the data type of each argument within the built-in function.

The following table outlines the meaning of each subscript character used in this appendix:

Legend for subscript indicators

Subscript	Data type
a	Any data type
a(non)	Any non-array
b	Boolean
c	Currency
f	Float
I	Integer
s	String
sa	Same as array
sm	Same as match array
p	Placeholder
n	Numeric

A.1 Array Functions

AdjustArray(Array $_{f}$[, Digits $_{i}$, Total $_{i}$])

Average(Array $_{i/f/c}$[, Start $_{i}$, Stop $_{i}$])

Contains(Array $_{a}$, Target $_{sa}$[, Start $_{i}$, Stop $_{i}$, Substring $_{b}$, NoCase $_{b}$], Sorted $_{b}$])

ContainsNot(Array $_{a}$, Target $_{sa}$[, Start $_{i}$, Stop $_{i}$, Substring $_{b}$, NoCase $_{b}$])

Count(Array $_{a}$)

Insert(Array $_{a}$, Index $_{i}$, Target $_{sa}$)

Join(Array $_{s}$[, Delimiter $_{s}$])

Max(Array $_{a}$[, Start $_{i}$, Stop $_{i}$])

Min(Array $_{a}$[, Start $_{i}$, Stop $_{i}$])

Remove(Array $_{a}$, Index $_{i}$)

Resize(Array $_{a}$, Index $_{i}$)

Sort(KeyArray $_{a}$, A/D $_{s}$, Array1 $_{a}$[, Array2 $_{a}$..., Array9 $_{a}$])

SortMulti(Array1 $_{a}$, A/D $_{s}$[, Array2 $_{a}$, A/D $_{s}$...])

SortRange(Start $_{i}$, End $_{i}$, Array1 $_{a}$, A/D $_{s}$[, Array2 $_{a}$...])

Split(Array $_{s}$, String $_{s}$[, Delimiter $_{s}$, Limit $_{i}$])

Sum(Array $_{i/f/c}$[, Start $_{i}$, Stop $_{i}$])

TrimArray(Array $_{a}$[, Values $_{a}$])

WhenAverage(Array $_{i/f/c}$, MatchArray $_{a}$, Target $_{sm}$[, SubString $_{b}$, NoCase $_{b}$])

WhenCount(Array $_{a}$, MatchArray $_{a}$, Target $_{sm}$[, SubString $_{b}$, NoCase $_{b}$])

WhenSum(Array $_{i/f/c}$, MatchArray $_{a}$, Target $_{sm}$, SubString $_{b}$, NoCase $_{b}$)

A.2 Data Section Functions

AddDataSection(DataSectionName $_{s}$, SiblingInstance $_{i}$, ParentName $_{s}$, ParentInstance $_{i}$, Before $_{b}$)

AddRecipient([RecipientProfileName $_{s}$, NumCopies $_{i}$])

CountDataSection(DataSectionName $_{s}$, InstanceNumber $_{i}$)

DeleteDataSection(DataSectionName $_{s}$, InstanceNumberOrIndex $_{i}$)

DuplicateDataSection(CopyTheData $_{b}$, Before $_{b}$, SelectNewDataSection $_{b}$)

MergeDataSection(DataSectionName $_{s}$, TargetInstanceNumber $_{i}$, InstanceNumber $_{i}$)

MoveDataSection(SiblingInstance $_{i}$, ParentName $_{s}$, ParentInstance $_{i}$, Before $_{b}$)

SelectDataSection(DataSectionName $_{s}$, InstanceNumberOrIndex $_{i}$, ChildNumber $_{i}$)

A.3 Date Functions

Date() - No arguments

DateAdd(Interval $_{s}$, Number of Periods $_{i/f}$, Original Date)

DateDiff(Interval $_{s}$, Date1, Date2[, FirstDayOfWeek $_{i}$, FirstWeekOfYear $_{i}$])

DatePart(Interval $_{s}$, Date[, FirstDayOfWeek $_{i}$, FirstWeekOfYear $_{i}$])

DateSerial(Year $_{i}$, Month $_{i}$, Day $_{i}$)

DateValue(DateString $_{s}$)

Day(Date)

IsDate(Expression $_{s}$)

Month(Date)

Weekday(Date[, FirstDayOfWeek $_{i}$])

Year(Date)

A.4 General Functions

ASCII(String $_{s}$[, ForceConversion $_{b}$])

Barcode(String $_{s}$, MapString $_{s}$[, BarcodeType $_{i}$])

CanImport(VariableName $_{p}$[,Index $_{i}$])

Choose(Index $_{i}$, Value1 $_{s}$, Value2 $_{s}$,...)

EBCDIC(String $_{s}$[, ForceConversion $_{b}$])

GetOS() - No arguments

InJurisdiction(JurisdictionName $_{s}$[, NameOrID $_{i}$])

IsEmpty(VariableName $_{a(non)}$)

IsNull(VariableName $_{a(non)}$)

IsNumeric(Expression $_{s}$)

LabelChar(FormatType $_{s}$)

Message(Number $_{i}$, Severity $_{s}$, Text $_{s}$[, MaxToShow $_{i}$, ExitRC $_{i}$, Error Level $_{i}$])

MessageCount(MessageType $_{s}$, CountType $_{i}$)

ResetVariable(VariableName $_s$)

RunMode() - No arguments

SetColorCMYK(ColorName $_s$, C-Value $_i$, M-Value $_i$, Y-Value $_i$, K-Value $_i$)

SetColorRGB(ColorName $_s$, R-Value $_i$, G-Value $_i$, B-Value $_i$)

Shell(Command $_s$[, SynchronousFlag $_b$])

StatusString(StatusCode $_i$)

Switch(Expr1 $_b$, Value1 $_a$[, Expr2 $_b$, Value2 $_a$,...])

A.5 I/O Functions

CanImport(VariableName $_p$[,Index $_i$])

DDAFileMap(DataFileName $_s$, Module $_s$, Routine $_s$, BufSize $_i$ [, Param $_s$, Type $_i$, Trigger $_b$])

DSNMap(DataFileName $_s$, DSN $_s$[, Username $_s$, Password $_s$, Schema $_s$, Trigger $_b$])

EvaluateXpath(Array$_a$, DataFileName$_s$, DefaultNamespace$_s$, XPathExpression$_s$, [AdditionalNamespaces$_s$])

FileMap(DataFileName $_s$, FileSystemFileName $_s$[, Trigger $_b$])

MapLayout(Buffer $_s$, DataFileName $_s$)

Trigger(DataFileName $_s$[, Function $_i$, SectionOrTagName $_s$])

WebServiceMap(DataFileName $_s$, URL $_s$[, SOAP Action $_s$, Options $_i$, Trigger $_b$])

A.6 Math Functions

Abs(Number $_{i/f/c}$)

Average(Array $_{i/f/c}$[, Start $_i$, Stop $_i$])

Contains(Array $_a$, Target $_{sa}$[, Start $_i$, Stop $_i$, Substring $_b$, NoCase $_b$], Sorted $_b$])

ContainsNot(Array $_a$, Target $_{sa}$[, Start $_i$, Stop $_i$, Substring $_b$, NoCase $_b$])

Count(Array $_a$)

Hex(Number $_i$)

Int(Number $_{f/c}$)

Max(Array $_a$[, Start $_i$, Stop $_i$])

Min(Array $_{a}$[, Start $_{i}$, Stop $_{i}$])

Rnd[(Number $_{n}$)]

Sum(Array $_{i/f/c}$[, Start $_{i}$, Stop $_{i}$])

WhenAverage(Array $_{i/f/c}$, MatchArray $_{a}$, Target $_{sm}$[, SubString $_{b}$, NoCase $_{b}$])

WhenCount(Array $_{a}$, MatchArray $_{a}$, Target $_{sm}$[, SubString $_{b}$, NoCase $_{b}$])

WhenSum(Array $_{i/f/c}$, MatchArray $_{a}$, Target $_{sm}$, SubString $_{b}$, NoCase $_{b}$)

A.7 String Functions

Asc(String $_{s}$)

ASCII(String $_{s}$[, ForceConversion $_{b}$])

Chr(Charcode $_{i}$)

EBCDIC(String $_{s}$[, ForceConversion $_{b}$])

Format(Expression $_{a}$[, Format $_{s}$, FirstDayOfWeek $_{i}$, FirstWeekOfYear $_{i}$])

Instr([Start $_{i}$,] String $_{s}$, Target $_{s}$)

Join(Array $_{s}$[, Delimiter $_{s}$])

LCase(String $_{s}$)

Left(String $_{s}$, Count $_{i}$)

Len(String $_{s}$)

LTrim(String $_{s}$)

MapLayout(Buffer $_{s}$, DataFileName $_{s}$)

Mid(String $_{s}$, Start $_{i}$[, Count $_{i}$])

MidWord(String $_{s}$, Start $_{i}$[, Count $_{i}$])

Replace(Strings, Find $_{s}$, Replace $_{s}$[, Start $_{i}$, Count $_{i}$])

Right(String $_{s}$, Count $_{i}$)

RTrim(String $_{s}$)

Space(Count $_{i}$)

Split(Array $_{s}$, String $_{s}$[, Delimiter $_{s}$, Limit $_{i}$])

StatusString(StatusCode $_{i}$)

Str(Number $_{i}$)

StrConv(String $_s$, Conversion $_i$)

String(Count $_i$, String $_s$)

Trim(String $_s$)

UCase(String $_s$)

Val(String $_s$)