INFORMATICA®

Informatica PowerCenter (Version 9.0.1)

Transformation Language Reference

Informatica PowerCenter Transformation Language Reference

Version 9.0.1
June 2010

DISCLAIMER: Informatica Corporation provides this documentation "as is" without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of non-infringement, merchantability, or use for a particular purpose. Informatica Corporation does not warrant that this software or documentation is error free. The information provided in this software or documentation may include technical inaccuracies or typographical errors. The information in this software and documentation is subject to change at any time without notice.

NOTICES

This Informatica product (the “Software”) includes certain drivers (the “DataDirect Drivers”) from DataDirect Technologies, an operating company of Progress Software Corporation (“DataDirect”) which are subject to the following terms and conditions:

1. THE DATADIRECT DRIVERS ARE PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT.
2. IN NO EVENT WILL DATADIRECT OR ITS THIRD PARTY SUPPLIERS BE LIABLE TO THE END-USER CUSTOMER FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL OR OTHER DAMAGES ARISING OUT OF THE USE OF THE ODBC DRIVERS, WHETHER OR NOT INFORMED OF THE POSSIBILITIES OF DAMAGES IN ADVANCE. THESE LIMITATIONS APPLY TO ALL CAUSES OF ACTION, INCLUDING, WITHOUT LIMITATION, BREACH OF CONTRACT, BREACH OF WARRANTY, NEGLIGENCE, STRICT LIABILITY, MISREPRESENTATION AND OTHER TORTS.

Part Number: PC-TLR-90100-0001

Table of Contents

Preface

The *PowerCenter Transformation Language Reference* is written for the developers who are responsible for building mappings. The *PowerCenter Transformation Language Reference* assumes you have knowledge of SQL, relational database concepts, and the interface requirements for your supporting applications.

Informatica Resources

Informatica Customer Portal

As an Informatica customer, you can access the Informatica Customer Portal site at http://mysupport.informatica.com. The site contains product information, user group information, newsletters, access to the Informatica customer support case management system (ATLAS), the Informatica How-To Library, the Informatica Knowledge Base, the Informatica Multimedia Knowledge Base, Informatica Product Documentation, and access to the Informatica user community.

Informatica Documentation

The Informatica Documentation team takes every effort to create accurate, usable documentation. If you have questions, comments, or ideas about this documentation, contact the Informatica Documentation team through email at infa_documentation@informatica.com. We will use your feedback to improve our documentation. Let us know if we can contact you regarding your comments.

The Documentation team updates documentation as needed. To get the latest documentation for your product, navigate to Product Documentation from http://mysupport.informatica.com.

Informatica Web Site

You can access the Informatica corporate web site at http://www.informatica.com. The site contains information about Informatica, its background, upcoming events, and sales offices. You will also find product and partner information. The services area of the site includes important information about technical support, training and education, and implementation services.

Informatica How-To Library

As an Informatica customer, you can access the Informatica How-To Library at http://mysupport.informatica.com. The How-To Library is a collection of resources to help you learn more about Informatica products and features. It includes articles and interactive demonstrations that provide solutions to common problems, compare features and behaviors, and guide you through performing specific real-world tasks.

Informatica Knowledge Base

As an Informatica customer, you can access the Informatica Knowledge Base at http://mysupport.informatica.com. Use the Knowledge Base to search for documented solutions to known technical issues about Informatica products. You can also find answers to frequently asked questions, technical white papers, and technical tips. If you have questions, comments, or ideas about the Knowledge Base, contact the Informatica Knowledge Base team through email at KB_Feedback@informatica.com.

Informatica Multimedia Knowledge Base

As an Informatica customer, you can access the Informatica Multimedia Knowledge Base at http://mysupport.informatica.com. The Multimedia Knowledge Base is a collection of instructional multimedia files that help you learn about common concepts and guide you through performing specific tasks. If you have questions, comments, or ideas about the Multimedia Knowledge Base, contact the Informatica Knowledge Base team through email at KB_Feedback@informatica.com.

Informatica Global Customer Support

You can contact a Customer Support Center by telephone or through the Online Support. Online Support requires a user name and password. You can request a user name and password at http://mysupport.informatica.com.

Use the following telephone numbers to contact Informatica Global Customer Support:

North America / South America	Europe / Middle East / Africa	Asia / Australia
Toll Free +1 877 463 2435 **Standard Rate** Brazil: +55 11 3523 7761 Mexico: +52 55 1168 9763 United States: +1 650 385 5800	**Toll Free** 00 800 4632 4357 **Standard Rate** Belgium: +32 15 281 702 France: +33 1 41 38 92 26 Germany: +49 1805 702 702 Netherlands: +31 306 022 797 Spain and Portugal: +34 93 480 3760 United Kingdom: +44 1628 511 445	**Toll Free** Australia: 1 800 151 830 Singapore: 001 800 4632 4357 **Standard Rate** India: +91 80 4112 5738

CHAPTER 1

The Transformation Language

This chapter includes the following topics:

The Transformation Language Overview

PowerCenter provides a transformation language that includes SQL-like functions to transform source data. Use these functions to write expressions and create functions called user-defined functions.

User-defined functions reuse expression logic and build complex expressions. You can include them in other user-defined functions or in expressions. User-defined functions follow the same guidelines as expressions. They use the same syntax and can use the same transformation language components.

Expressions modify data or test whether data matches conditions. For example, you might use the AVG function to calculate the average salary of all the employees, or the SUM function to calculate the total sales for a specific branch.

You can create a simple expression that only contains a port, such as ORDERS, or a numeric literal, such as 10. You can also write complex expressions that include functions nested within functions, or combine different ports using the transformation language operators. For more information about how transformation expressions are evaluated, see “Working with Transformations” in the *PowerCenter Transformation Guide*.

Transformation Language Components

The transformation language includes the following components to create simple or complex transformation expressions:

- **Functions.** Over 100 SQL-like functions allow you to change data in a mapping.
- **Operators.** Use transformation operators to create transformation expressions to perform mathematical computations, combine data, or compare data.
- **Constants.** Use built-in constants to reference values that remain constant, such as TRUE.
- **Mapping parameters and variables.** Create mapping parameters for use within a mapping or mapplet to reference values that remain constant throughout a session, such as a state sales tax rate. Create mapping variables in mapplets or mappings to write expressions referencing values that change from session to session.

- **Workflow variables.** Create workflow variables for use within a workflow to write expressions referencing values that change from workflow to workflow.
- **Built-in and local variables.** Use built-in variables to write expressions that reference values that vary, such as the system date. You can also create local variables in transformations.
- **Return values.** You can also write expressions that include the return values from Lookup, Stored Procedure, and External Procedure transformations.

Internationalization and the Transformation Language

Transformation language functions can handle character data in either ASCII or Unicode data movement mode. Use Unicode mode to handle *multibyte* character data. The return values of the following functions and transformations depend on the code page of the Integration Service and the data movement mode:

- INITCAP
- LOWER
- UPPER
- MIN (Date)
- MIN (Number)
- MIN (String)
- MAX (Date)
- MAX (Number)
- MAX (String)
- Any function that uses conditional statements to compare strings, such as IIF and DECODE

MIN and MAX also return values based on the sort order associated with the Integration Service code page.

When you validate an invalid expression in the Expression Editor, a dialog box displays the expression with an error indicator, “>>>>”. This indicator appears to the left of and points to the part of the expression containing the error. For example, if the expression a = b + c contains an error at c, the error message displays:

```
a = b +   >>>> c
```

Transformation language functions that evaluate character data are character-oriented, not byte-oriented. For example, the LENGTH function returns the number of characters in a string, not the number of bytes. The LOWER function returns a string in lowercase based on the code page of the Integration Service.

Expression Syntax

Although the transformation language is based on standard SQL, there are difference between the two languages. For example, SQL supports the keywords ALL and DISTINCT for aggregate functions, but the transformation language does not. On the other hand, the transformation language supports an optional filter condition for aggregate functions, while SQL does not.

You can create an expression that is as simple as a port (such as ORDERS), a pre-defined workflow variable (such as $Start.Status), or a numeric literal (such as 10). You can also write complex expressions that include functions nested within functions, or combine different columns using the transformation language operators.

Expression Components

Expressions can consist of any combination of the following components:

- Ports (input, input/output, variable)
- String literals, numeric literals
- Constants
- Functions
- Built-in and local variables
- Mapping parameters and mapping variables
- Pre-defined workflow variables
- User-defined workflow variables
- Operators
- Return values

Ports and Return Values

When you write an expression that includes a port or return value from an unconnected transformation, use the reference qualifiers in the following table:

Reference Qualifier	Description
:EXT	Required when you write an expression that includes a return value from an External Procedure transformation. The general syntax is: `:EXT.external_procedure_transformation(argument1, argument2, ...)`
:LKP	Required when you create an expression that includes the return value from an unconnected Lookup transformation. The general syntax is: `:LKP.lookup_transformation(argument1, argument2, ...)` The arguments are the local ports used in the lookup condition. The order must match the order of the ports in the transformation. The datatypes for the local ports must match the datatype of the Lookup ports used in the lookup condition.
:SD	Optional (PowerMart 3.5 expressions only). Qualifies a source table port in an expression. The general syntax is: `:SD.source_table.column_name`
:SEQ	Required when you create an expression that includes a port in a Sequence Generator transformation. The general syntax is: `:SEQ.sequence_generator_transformation.CURRVAL`
:SP	Required when you write an expression that includes the return value from an unconnected Stored Procedure transformation. The general syntax is: `:SP.stored_procedure_transformation( argument1, argument2, [, PROC_RESULT])` The arguments must match the arguments in the unconnected Stored Procedure transformation.
:TD	Required when you reference a target table in a PowerMart 3.5 LOOKUP function. The general syntax is: `LOOKUP(:TD.SALES.ITEM_NAME, :TD.SALES.ITEM_ID, 10, :TD.SALES.PRICE, 15.99)`

String and Numeric Literals

You can include numeric or string literals.

Be sure to enclose string literals within single quotation marks. For example:

```
'Alice Davis'
```

String literals are case sensitive and can contain any character except a single quotation mark. For example, the following string is not allowed:

```
'Joan's car'
```

To return a string containing a single quote, use the CHR function:

```
'Joan' || CHR(39) || 's car'
```

Do not use single quotation marks with numeric literals. Just enter the number you want to include. For example:

```
.05
```

or

```
$$Sales_Tax
```

Rules and Guidelines for Expression Syntax

Use the following rules and guidelines when you write expressions:

- You cannot include both single-level and nested aggregate functions in an Aggregator transformation.
- If you need to create both single-level and nested functions, create separate Aggregator transformations.
- You cannot use strings in numeric expressions.

  For example, the expression `1 + '1'` is not valid because you can only perform addition on numeric datatypes. You cannot add an integer and a string.
- You cannot use strings as numeric parameters.

  For example, the expression `SUBSTR(TEXT_VAL, '1', 10)` is not valid because the SUBSTR function requires an integer value, not a string, as the start position.
- You cannot mix datatypes when using comparison operators.

  For example, the expression `123.4 = '123.4'` is not valid because it compares a decimal value with a string.
- You can pass a value from a port, literal string or number, variable, Lookup transformation, Stored Procedure transformation, External Procedure transformation, or the results of another expression.
- Use the ports tab in the Expression Editor to enter a port name into an expression. If you rename a port in a connected transformation, the Designer propagates the name change to expressions in the transformation.
- Separate each argument in a function with a comma.
- Except for literals, the transformation language is not case sensitive.
- Except for literals, the Designer and Integration Service ignore spaces.
- The colon (:), comma (,), and period (.) have special meaning and should be used only to specify syntax.
- The Integration Service treats a dash (-) as a minus operator.
- If you pass a literal value to a function, enclose literal strings within single quotation marks. Do not use quotation marks for literal numbers. The Integration Service treats any string value enclosed in single quotation marks as a character string.
- When you pass a mapping parameter or variable or a workflow variable to a function within an expression, do not use quotation marks to designate mapping parameters or variables or workflow variables.
- Do not use quotation marks to designate ports.
- You can nest multiple functions within an expression (except aggregate functions, which allow only one nested aggregate function). The Integration Service evaluates the expression starting with the innermost function.

Adding Comments to Expressions

The transformation language provides two comment specifiers to let you insert comments in expressions:

- Two dashes, as in:

```
-- These are comments
```

- Two slashes, as in:

```
// These are comments
```

The Integration Service ignores all text on a line preceded by these two comment specifiers. For example, if you want to concatenate two strings, you can enter the following expression with comments in the middle of the expression:

```
 -- This expression concatenates first and last names for customers:
FIRST_NAME -- First names from the CUST table
|| // Concat symbol
LAST_NAME // Last names from the CUST table
// Joe Smith Aug 18 1998
```

The Integration Service ignores the comments and evaluates the expression as follows:

```
FIRST_NAME || LAST_NAME
```

You cannot continue a comment to a new line:

```
-- This expression concatenates first and last names for customers:
FIRST_NAME -- First names from the CUST table
|| // Concat symbol
LAST_NAME // Last names from the CUST table
Joe Smith Aug 18 1998
```

In this case, the Designer and Workflow Manager do not validate the expression, since the last line is not a valid expression.

If you do not want to embed comments, you can add them by clicking Comment in the Expression Editor.

Reserved Words

Some keywords in the transformation language, such as constants, operators, and built-in variables, are reserved for specific functions. These include:

- :EXT
- :INFA
- :LKP
- :MCR
- :SD
- :SEQ
- :SP
- :TD
- AND
- DD_DELETE
- DD_INSERT
- DD_REJECT
- DD_UPDATE

- FALSE
- NOT
- NULL
- OR
- PROC_RESULT
- SESSSTARTTIME
- SPOUTPUT
- SYSDATE
- TRUE
- WORKFLOWSTARTTIME

The following words are reserved for workflow expressions:

- ABORTED
- DISABLED
- FAILED
- NOTSTARTED
- STARTED
- STOPPED
- SUCCEEDED

Note: You cannot use a reserved word to name a port or local variable. You can only use reserved words within transformation and workflow expressions. Reserved words have predefined meanings in expressions.

CHAPTER 2

Constants

This chapter includes the following topics:

DD_DELETE

Flags records for deletion in an update strategy expression. DD_DELETE is equivalent to the integer literal 2.

Note: Use the DD_DELETE constant in the Update Strategy transformation only. Use DD_DELETE instead of the integer literal 2 to facilitate troubleshooting complex numeric expressions.

When you run a workflow, select the data-driven update strategy to delete records from a target based on this flag.

Example

The following expression marks items with an ID number of 1001 for deletion, and all other items for insertion:

```
IIF( ITEM_ID = 1001, DD_DELETE, DD_INSERT )
```

This update strategy expression uses numeric literals to produce the same result:

```
IIF( ITEM_ID = 1001, 2, 0 )
```

Note: The expression using constants is easier to read than the expression using numeric literals.

DD_INSERT

Flags records for insertion in an update strategy expression. DD_INSERT is equivalent to the integer literal 0.

Note: Use the DD_INSERT constant in the Update Strategy transformation only. Use DD_INSERT instead of the integer literal 0 to facilitate troubleshooting complex numeric expressions.

When you run a workflow, select the data-driven update strategy to write records to a target based on this flag.

Examples

The following examples modify a mapping that calculates monthly sales by salesperson, so you can examine the sales of just one salesperson.

The following update strategy expression flags an employee's sales for insertion, and rejects everything else:

```
IIF( EMPLOYEE.NAME = 'Alex', DD_INSERT, DD_REJECT )
```

This update strategy expression uses numeric literals to produce the same result:

```
IIF( EMPLOYEE.NAME = 'Alex', 0, 3 )
```

Tip: The expression using constants is easier to read than the expression using numeric literals.

The following update strategy expression uses SESSSTARTTIME to find only those orders that shipped in the last two days and flag them for insertion. Using DATE_DIFF, the expression subtracts DATE_SHIPPED from the system date, returning the difference between the two dates. Because DATE_DIFF returns a Double value, the expression uses TRUNC to truncate the difference. It then compares the result to the integer literal 2. If the result is greater than 2, the expression flags the records for rejection. If the result is 2 or less, it flags them for insertion:

```
IIF( TRUNC( DATE_DIFF( SESSSTARTTIME, ORDERS.DATE_SHIPPED, 'DD' ), 0 ) > 2,DD_REJECT, DD_INSERT )
```

DD_REJECT

Flags records for rejection in an update strategy expression. DD_REJECT is equivalent to the integer literal 3.

Note: Use the DD_REJECT constant in the Update Strategy transformation only. Use DD_REJECT instead of the integer literal 3 to facilitate troubleshooting complex numeric expressions.

When you run a workflow, select the data-driven update strategy to reject records from a target based on this flag.

Use DD_REJECT to filter or validate data. If you flag a record as reject, the Integration Service skips the record and writes it to the session reject file.

Examples

The following examples modify a mapping that calculates the sales for the current month, so it includes only positive values.

This update strategy expression flags records less than 0 for reject and all others for insert:

```
IIF( ORDERS.SALES > 0, DD_INSERT, DD_REJECT )
```

This expression uses numeric literals to produce the same result:

```
IIF( ORDERS.SALES > 0, 0, 3 )
```

Notice that the expression using constants is easier to read than the expression using numeric literals.

The following data-driven example uses DD_REJECT and IS_SPACES to avoid writing spaces to a character column in a target table. This expression flags records that consist entirely of spaces for reject and flags all others for insert:

```
IIF( IS_SPACES( CUST_NAMES ), DD_REJECT, DD_INSERT )
```

DD_UPDATE

Flags records for update in an update strategy expression. DD_UPDATE is equivalent to the integer literal 1.

Note: Use the DD_UPDATE constant in the Update Strategy transformation only. Use DD_UPDATE instead of the integer literal 1 to facilitate troubleshooting complex numeric expressions.

When you run a workflow, select the data-driven update strategy to write records to a target based on this flag.

Examples

The following examples modify a mapping that calculates sales for the current month. The mapping loads sales for one employee.

This expression flags records for Alex as updates and flags all others for rejection:

```
IIF( EMPLOYEE.NAME = 'Alex', DD_UPDATE, DD_REJECT )
```

This expression uses numeric literals to produce the same result, flagging Alex's sales for update (1) and flagging all other sales records for rejection (3):

```
IIF( EMPLOYEE.NAME = 'Alex', 1, 3 )
```

Notice that the expression using constants is easier to read than the expression using numeric literals.

The following update strategy expression uses SYSDATE to find only those orders that have shipped in the last two days and flag them for insertion. Using DATE_DIFF, the expression subtracts DATE_SHIPPED from the system date, returning the difference between the two dates. Because DATE_DIFF returns a Double value, the expression uses TRUNC to truncate the difference. It then compares the result to the integer literal 2. If the result is greater than 2, the expression flags the records for rejection. If the result is 2 or less, it flags the records for update. Otherwise, it flags them for rejection:

```
IIF( TRUNC( DATE_DIFF( SYSDATE, ORDERS.DATE_SHIPPED, 'DD' ), 0 ) > 2, DD_REJECT, DD_UPDATE )
```

FALSE

Clarifies a conditional expression. FALSE is equivalent to the integer 0.

Example

The following example uses FALSE in a DECODE expression to return values based on the results of a comparison. This is useful if you want to perform multiple searches based on a single search value:

```
DECODE( FALSE,
Var1 = 22,'Variable 1 was 22!',
Var2 = 49,'Variable 2 was 49!',
Var1 < 23, 'Variable 1 was less than 23.',
Var2 > 30, 'Variable 2 was more than 30.',
'Variables were out of desired ranges.')
```

NULL

Indicates that a value is either unknown or undefined. NULL is not equivalent to a blank or empty string (for character columns) or 0 (for numerical columns).

Although you can write expressions that return nulls, any column that has the NOT NULL or PRIMARY KEY constraint will not accept nulls. Therefore, if the Integration Service tries to write a null value to a column with one of these constraints, the database will reject the row and the Integration Service will write it to the reject file. Be sure to consider nulls when you create transformations.

Functions can handle nulls differently. If you pass a null value to a function, it might return 0 or NULL, or it might ignore null values.

Related Topics:

- "Functions" on page 34

Working with Null Values in Boolean Expressions

Expressions that combine a null value with a Boolean expression produces results that are ANSI compliant. For example, the Integration Service produces the following results:

- NULL AND TRUE = NULL
- NULL AND FALSE = FALSE

Working with Null Values in Comparison Expressions

By default, when you use a null value in an expression containing a comparison operator, the Integration Service produces a null value. However, you can also configure the Integration Service to treat null values as high or low in comparison operations.

Use the Treat Null In Comparison Operators As property to configure how the Integration Service handles null values in comparison expressions.

This Integration Service configuration property affects the behavior of the following comparison operators in expressions:

```
=, !=, ^=, <>, >, >=, <, <=
```

For example, consider the following expressions:

```
NULL > 1
NULL = NULL
```

The following table describes how the Integration Service evaluates the expressions:

Expression	Treat Null in Comparison Operators As		
	NULL	HIGH	LOW
NULL > 1	NULL	TRUE	FALSE
NULL = NULL	NULL	TRUE	TRUE

Null Values in Aggregate Functions

By default, the Integration Service treats null values as nulls in aggregate functions. If you pass an entire port or group of null values, the function returns NULL. However, when you configure the Integration Service, you can choose how you want it to handle null values in aggregate functions. You can have the Integration Service treat null values as 0 in aggregate functions or as NULLs.

Null Values in Filter Conditions

If a filter condition evaluates to NULL, the function does not select the record. If the filter condition evaluates to NULL for all records in the selected port, the aggregate function returns NULL (except COUNT, which returns 0). You use filter conditions with aggregate functions, and these functions CUME, MOVINGAVG, and MOVINGSUM.

Nulls with Operators

Any expression that uses operators (except the string operator ||) and contains a null value always evaluates to NULL. For example, the following expression evaluates to NULL:

```
8 * 10 - NULL
```

To test for nulls, use the ISNULL function.

TRUE

Returns a value based on the result of a comparison. TRUE is equivalent to the integer 1.

Example

The following example uses TRUE in a DECODE expression to return values based on the results of a comparison. This is useful if you want to perform multiple searches based on a single search value:

```
DECODE( TRUE,
Var1 = 22,'Variable 1 was 22!',
Var2 = 49,'Variable 2 was 49!',
Var1 < 23, 'Variable 1 was less than 23.',
Var2 > 30, 'Variable 2 was more than 30.',
'Variables were out of desired ranges.')
```

CHAPTER 3

Operators

This chapter includes the following topics:

Operator Precedence

The transformation language supports the use of multiple operators and the use of operators within nested expressions.

If you write an expression that includes multiple operators, the Integration Service evaluates the expression in the following order:

1. Arithmetic operators
2. String operators
3. Comparison operators
4. Logical operators

The Integration Service evaluates operators in the order they appear in the following table. It evaluates operators in an expression with equal precedence to all operators from left to right.

The following table lists the precedence for all transformation language operators:

Operator	Meaning
( )	Parentheses.
+, -, NOT	Unary plus and minus and the logical NOT operator.
*, /,%	Multiplication, division, modulus.
+, -	Addition, subtraction.
||	Concatenate.

Operator	Meaning
<, <=, >, >=	Less than, less than or equal to, greater than, greater than or equal to.
=, <>, !=, ^=	Equal to, not equal to, not equal to, not equal to.
AND	Logical AND operator, used when specifying conditions.
OR	Logical OR operator, used when specifying conditions.

The transformation language also supports the use of operators within nested expressions. When expressions contain parentheses, the Integration Service evaluates operations inside parentheses before operations outside parentheses. Operations in the innermost parentheses are evaluated first.

For example, depending on how you nest the operations, the equation 8 + 5 - 2 * 8 returns different values:

Equation	Return Value
8 + 5 - 2 * 8	-3
8 + (5 - 2) * 8	32

Arithmetic Operators

Use arithmetic operators to perform mathematical calculations on numeric data.

The following table lists the arithmetic operators in order of precedence in the transformation language:

Operator	Meaning
+, -	Unary plus and minus. Unary plus indicates a positive value. Unary minus indicates a negative value.
*, /, %	Multiplication, division, modulus. A modulus is the remainder after dividing two integers. For example, 13 % 2 = 1 because 13 divided by 2 equals 6 with a remainder of 1.
+, -	Addition, subtraction. The addition operator (+) does not concatenate strings. To concatenate strings, use the string operator ||. To perform arithmetic on date values, use the date functions.

If you perform arithmetic on a null value, the function returns NULL.

When you use arithmetic operators in an expression, all of the operands in the expression must be numeric. For example, the expression `1 + '1'` is not valid because it adds an integer to a string. The expression `1.23 + 4 / 2` is valid because all of the operands are numeric.

Note: The transformation language provides built-in date functions that let you perform arithmetic on date/time values.

Related Topics:

- “Understanding Date Arithmetic” on page 33

String Operators

Use the || string operator to concatenate two strings. The || operator converts operands of any datatype (except Binary) to String datatypes before concatenation:

Input Value	Return Value
'alpha' || 'betical'	alphabetical
'alpha' || 2	alpha2
'alpha' || NULL	alpha

The || operator includes leading and trailing blanks. Use the LTRIM and RTRIM functions to trim leading and trailing blanks before concatenating two strings.

Nulls

The || operator ignores null values. However, if both values are NULL, the || operator returns NULL.

Example

The following example shows an expression that concatenates employee first names and employee last names from two columns. This expression removes the spaces from the end of the first name and the beginning of the last name, concatenates a space to the end of each first name, then concatenates the last name:

```
LTRIM( RTRIM( EMP_FIRST ) || ' ' || LTRIM( EMP_LAST ))
```

```
EMP_FIRST          EMP_LAST           RETURN VALUE
' Alfred'          ' Rice '           Alfred Rice
' Bernice'         ' Kersins'         Bernice Kersins
NULL               ' Proud'           Proud
' Curt'            NULL               Curt
NULL               NULL               NULL
```

Note: You can also use the CONCAT function to concatenate two string values. The || operator, however, produces the same results in less time.

Comparison Operators

Use comparison operators to compare character or numeric strings, manipulate data, and return a TRUE (1) or FALSE (0) value.

The following table lists the comparison operators in the transformation language:

Operator	Meaning
=	Equal to.
>	Greater than.
<	Less than.
>=	Greater than or equal to.
<=	Less than or equal to.
<>	Not equal to.
!=	Not equal to.
^=	Not equal to.

Use the greater than (>) and less than (<) operators to compare numeric values or return a range of rows based on the sort order for a primary key in a particular port.

When you use comparison operators in an expression, the operands must be the same datatype. For example, the expression 123.4 > '123' is not valid because the expression compares a decimal with a string. The expressions 123.4 > 123 and 'a' != 'b' are valid because the operands are the same datatype.

If you compare a value to a null value, the result is NULL.

If a filter condition evaluates to NULL, the Integration Service returns NULL.

Logical Operators

Use logical operators to manipulate numeric data. Expressions that return a numeric value evaluate to TRUE for values other than 0, FALSE for 0, and NULL for NULL.

The following table lists the logical operators in the transformation language:

Operator	Meaning
NOT	Negates result of an expression. For example, if an expression evaluates to TRUE, the operator NOT returns FALSE. If an expression evaluates to FALSE, NOT returns TRUE.
AND	Joins two conditions and returns TRUE if both conditions evaluate to TRUE. Returns FALSE if one condition is not true.
OR	Connects two conditions and returns TRUE if any condition evaluates to TRUE. Returns FALSE if both conditions are not true.

Nulls

Expressions that combine a null value with a Boolean expression produce results that are ANSI compliant. For example, the Integration Service produces the following results:

- NULL AND TRUE = NULL
- NULL AND FALSE = FALSE

CHAPTER 4

Variables

This chapter includes the following topics:

Built-in Variables

The transformation language provides built-in variables. Built-in variables return either run-time or system information. Run-time variables return information such as source and target table name, folder name, session run mode, and workflow run instance name. System variables return session start time, system date, and workflow start time.

You can use built-in variables in expressions in the Designer or Workflow Manager. For example, you can use the system variable SYSDATE in a DATE_DIFF function. You can use run-time variables in expressions and in input fields that accept mapping or workflow variables. For example, you can use run-time variable $PMWorkflowRunInstanceName as part of a target output file name. The Integration Service sets the values of built-in variables. You cannot define values for built-in variables in a workflow or session parameter file.

The following built-in variables provide run-time information:

- $PM<SourceName>@TableName, $PM<TargetName>@TableName
- $PMFolderName
- $PMIntegrationServiceName
- $PMMappingName
- $PMRepositoryServiceName
- $PMRepositoryUserName
- $PMSessionName
- $PMSessionRunMode
- $PMWorkflowName
- $PMWorkflowRunId
- $PMWorkflowRunInstanceName

The following built-in variables provide system information:

- $$$SessStartTime
- SESSSTARTTIME

- SYSDATE
- WORKFLOWSTARTTIME

The following table describes where you use built-in variables in the Designer and Workflow Manager:

Variable Name	Designer	Workflow Manager
$PM<SourceName>@TableName, $PM<TargetName>@TableName,	- Expressions - Input fields that accept mapping variables	- Input fields that accept mapping variables
$PMFolderName	- Expressions - Input fields that accept mapping variables - Input fields that accept workflow variables	- Expressions - Input fields that accept mapping variables - Input fields that accept workflow variables
$PMIntegrationServiceName	- Expressions - Input fields that accept mapping variables - Input fields that accept workflow variables	- Expressions - Input fields that accept mapping variables - Input fields that accept workflow variables
$PMMappingName	- Expressions - Input fields that accept mapping variables	- Input fields that accept mapping variables
$PMRepositoryServiceName	- Expressions - Input fields that accept mapping variables - Input fields that accept workflow variables	- Expressions - Input fields that accept mapping variables - Input fields that accept workflow variables
$PMRepositoryUserName	- Expressions - Input fields that accept mapping variables - Input fields that accept workflow variables	- Expressions - Input fields that accept mapping variables - Input fields that accept workflow variables
$PMSessionName	- Expressions - Input fields that accept mapping variables	- Input fields that accept mapping variables
$PMSessionRunMode	- Expressions - Input fields that accept mapping variables	- Input fields that accept mapping variables
$PMWorkflowName	- Expressions - Input fields that accept mapping variables - Input fields that accept workflow variables	- Expressions - Input fields that accept mapping variables - Input fields that accept workflow variables
$PMWorkflowRunId	- Expressions - Input fields that accept mapping variables - Input fields that accept workflow variables	- Expressions - Input fields that accept mapping variables - Input fields that accept workflow variables
$PMWorkflowRunInstanceName	- Expressions - Input fields that accept mapping variables - Input fields that accept workflow variables	- Expressions - Input fields that accept mapping variables - Input fields that accept workflow variables
$$$SessStartTime	- Mapping or mapplet filter conditions - User-defined joins - SQL overrides	- Mapping or mapplet filter conditions - User-defined joins - SQL overrides
SESSSTARTTIME	- Expressions	n/a
SYSDATE	- Expressions	- Expressions
WORKFLOWSTARTTIME	n/a	- Expressions

$PM<SourceName>@TableName, $PM<TargetName>@TableName

$PM<*SourceName*>@TableName and $PM<*TargetName*>@TableName return the source and target table names for relational source and target instances as string values. Use these variables with any function that accepts string datatypes.

The variable name depends on the source or target instance name. For example, for a source instance named “Customers,” the built-in variable name is $PMCustomers@TableName. If the relational source or target is part of a mapplet within a mapping, the built-in variable name includes the mapplet name:

- $PM<*MappletName*>.<*SourceName*>@TableName
- $PM<*MappletName*>.<*TargetName*>@TableName

Use $PM<*SourceName*>@TableName and $PM<*TargetName*>@TableName in a mapping or a mapplet. For example, in a mapping that contains multiple relational sources, you can use $PM<*SourceName*>@TableName in the output port of an Expression transformation to write the source table name for each row to the target. You can also use these variables in input fields that accept mapping variables.

$PMFolderName

$PMFolderName returns the name of the repository folder as a string value. Use $PMFolderName with any function that accepts string datatypes.

Use $PMFolderName in a mapping, a mapplet, workflow links, or in workflow tasks such as Assignment and Decision tasks. You can also use $PMFolderName in input fields that accept mapping or workflow variables.

$PMIntegrationServiceName

$PMIntegrationServiceName returns the name of the Integration Service that runs the session. Use $PMIntegrationServiceName with any function that accepts string datatypes. $PMIntegrationServiceName returns the Integration Service name as a string value.

Use $PMIntegrationServiceName in a mapping, a mapplet, workflow links, or in workflow tasks such as Assignment and Decision tasks. You can also use $PMIntegrationServiceName in input fields that accept mapping or workflow variables.

$PMMappingName

$PMMappingName returns the mapping name as a string value. Use $PMMappingName with any function that accepts string datatypes.

Use $PMMappingName in a mapping or a mapplet. You can also use $PMMappingName in input fields that accept mapping variables.

$PMRepositoryServiceName

$PMRepositoryServiceName returns the name of the Repository Service as a string value. Use $PMRepositoryServiceName with any function that accepts string datatypes.

Use $PMRepositoryServiceName in a mapping, a mapplet, workflow links, or in workflow tasks such as Assignment and Decision tasks. You can also use $PMRepositoryServiceName in input fields that accept mapping or workflow variables.

$PMRepositoryUserName

$PMRepositoryUserName returns the name of the repository user that runs the session. Use $PMRepositoryUserName with any function that accepts string datatypes. $PMRepositoryUserName returns the repository user name as a string value.

Use $PMRepositoryUserName in a mapping, a mapplet, workflow links, or in workflow tasks such as Assignment and Decision tasks. You can also use $PMRepositoryUserName in input fields that accept mapping or workflow variables.

$PMSessionName

$PMSessionName returns the session name as a string value. Use $PMSessionName with any function that accepts string datatypes.

Use $PMSessionName in a mapping or a mapplet. You can also use $PMSessionName in input fields that accept mapping variables.

$PMSessionRunMode

$PMSessionRunMode returns the session run mode, *normal* or *recovery*, as a string value. Use $PMSessionRunMode with any function that accepts string datatypes.

Use $PMSessionRunMode in a mapping or a mapplet. You can also use $PMSessionRunMode in input fields that accept mapping variables.

$PMWorkflowName

$PMWorkflowName returns the name of the workflow as a string value. Use $PMWorkflowName with any function that accepts string datatypes.

Use $PMWorkflowName in a mapping, a mapplet, workflow links, or in workflow tasks such as Assignment and Decision tasks. You can also use $PMWorkflowName in input fields that accept mapping or workflow variables.

$PMWorkflowRunId

Each workflow run has a unique run ID. $PMWorkflowRunId returns the workflow run ID as a string value. Use $PMWorkflowRunId with any function that accepts string datatypes.

Use $PMWorkflowRunId in a mapping, a mapplet, workflow links, or in workflow tasks such as Assignment and Decision tasks. You can also use $PMWorkflowRunId in input fields that accept mapping or workflow variables. For example, you configure a workflow to run concurrently with the same instance name, and you want to track the status of each workflow run using a third-party application. Use $PMWorkflowRunId in a post-session shell command to pass the run ID to the application.

$PMWorkflowRunInstanceName

$PMWorkflowRunInstanceName returns the workflow run instance name as a string value. Use $PMWorkflowRunInstanceName with any function that accepts string datatypes.

Use $PMWorkflowRunInstanceName in a mapping, a mapplet, workflow links, or in workflow tasks such as Assignment and Decision tasks. You can also use $PMWorkflowRunInstanceName in input fields that accept mapping or workflow variables. For example, for a concurrent workflow with unique instance names, you can create unique target files for each run instance by setting the target output file name in the session properties to “OutFile_$PMWorkflowRunInstanceName.txt.”

Or, you want to use a post-session shell command to create an indicator file used by a predefined Event-Wait task. In the shell command that generates the indicator file, use $PMWorkflowRunInstanceName in the indicator file name to ensure that one workflow run instance does not delete an indicator file needed by another workflow run instance.

SESSSTARTTIME

SESSSTARTTIME returns the current date and time value on the node that runs the session when the Integration Service initializes the session. Use SESSSTARTTIME with any function that accepts transformation date/time datatypes. SESSSTARTTIME is stored as a transformation date/time datatype value.

Use SESSSTARTTIME in a mapping or a mapplet. You can reference SESSSTARTTIME only within the expression language.

Example

The following expression uses $$$SessStartTime in the source filter condition of a source qualifier to perform an incremental extraction. The expression specifies a range of dates of all days in the week prior to when the Integration Service initializes the session. The expression uses the function DATE_DIFF to find the difference in the number of days between the value ORDER_DATE and $$$SessStartTime. If the difference between the two dates is less than or equal to seven days, the Integration Service extracts that row from the source:

```
DATE_DIFF(DAY, ORDER_DATE, '$$$SessStartTime') <= 7
```

SYSDATE

SYSDATE returns the current date and time up to seconds on the node that runs the session for each row passing through the transformation. SYSDATE is stored as a transformation date/time datatype value.

To capture a static system date, use the SESSSTARTTIME variable instead of SYSDATE.

Example

The following expression uses SYSDATE to find orders that have shipped in the last two days and flag them for insertion. Using DATE_DIFF, the Integration Service subtracts DATE_SHIPPED from the system date, returning the difference between the two dates. Because DATE_DIFF returns a double value, the expression truncates the difference. It then compares the result to the integer literal 2. If the result is greater than 2, the expression flags the rows for rejection. If the result is 2 or less, it flags them for insertion.

```
IIF( TRUNC( DATE_DIFF( SYSDATE, DATE_SHIPPED, 'DD' ),

0 ) > 2, DD_REJECT, DD_INSERT
```

WORKFLOWSTARTTIME

WORKFLOWSTARTTIME returns the current date and time on the node hosting the Integration Service when the Integration Service initializes the workflow. Use WORKFLOWSTARTTIME with any function that accepts transformation date/time datatypes. WORKFLOWSTARTTIME is stored as a transformation date/time datatype value.

Use WORKFLOWSTARTTIME in workflow links and tasks such as Assignment and Decision tasks. You can reference WORKFLOWSTARTTIME only within the expression language.

Example

The following expression uses WORKFLOWSTARTTIME to display the number of minutes between the workflow start time and the start time of a task in the workflow. Using the SQL function DATE_DIFF, the Integration Service subtracts the task start time from WORKFLOWSTARTTIME and returns the result as a number of days:

```
DATE_DIFF(WORKFLOWSTARTTIME, $s_EmployeeData.StartTime, 'MI')
```

Transaction Control Variables

Transaction control variables define conditions to commit or rollback transactions during the processing of database rows. You use these variables in transaction control expressions that you build in the Expression Editor. Transaction control expressions use the IIF function to test each row against a condition. Depending on the return value of the condition, the Integration Service commits, rolls back, or makes no transaction changes for the row.

The following example uses transaction control variables to determine where to process a row:

```
IIF (NEWTRAN=1, TC_COMMIT_BEFORE, TC_CONTINUE_TRANSACTION)
```

If NEWTRAN=1, the TC_COMMIT_BEFORE variable causes a commit to occur before the current row processes. Otherwise, the TC_CONTINUE_TRANSACTION variable forces the row to process in the current transaction.

Use the following variables in the Expression Editor when you create a transaction control expression:

- **TC_CONTINUE_TRANSACTION.** The Integration Service does not perform any transaction change for the current row. This is the default transaction control variable value.
- **TC_COMMIT_BEFORE.** The Integration Service commits the transaction, begins a new transaction, and writes the current row to the target. The current row is in the new transaction.
- **TC_COMMIT_AFTER.** The Integration Service writes the current row to the target, commits the transaction, and begins a new transaction. The current row is in the committed transaction.
- **TC_ROLLBACK_BEFORE.** The Integration Service rolls back the current transaction, begins a new transaction, and writes the current row to the target. The current row is in the new transaction.

Local Variables

If you use local variables in a mapping, use them in any transformation expression in the mapping. For example, if you use a complex tax calculation throughout a mapping, you might want to write the expression once and designate it as a variable. This increases performance since the Integration Service performs the calculation only once.

Local variables are useful when used with stored procedure expressions to capture multiple return values.

CHAPTER 5

Dates

This chapter includes the following topics:

Dates Overview

The transformation language provides a set of date functions and built-in date variables to perform transformations on dates. With the date functions, you can round, truncate, or compare dates, extract one part of a date, or perform arithmetic on a date. You can pass any value with a date datatype to a date function.

Use date variables to capture the current date or session start time on the node hosting the Integration Service.

The transformation language also provides the following sets of format strings:

- **Date format strings.** Use with date functions to specify the parts of a date.
- **TO_CHAR format strings.** Use to specify the format of the return string.
- **TO_DATE and IS_DATE format strings.** Use to specify the format of a string you want to convert to a date or test.

Date/Time Datatype

PowerCenter provides a set of generic datatypes to transform data from different sources. These transformation datatypes include a Date/Time datatype that supports datetime values up to the nanosecond. PowerCenter stores dates internally in binary format.

Date functions accept datetime values only. To pass a string to a date function, first use TO_DATE to convert it to a datetime value. For example, the following expression converts a string port to datetime values and then adds one month to each date:

```
ADD_TO_DATE( TO_DATE( STRING_PORT, 'MM/DD/RR'), 'MM', 1 )
```

PowerCenter supports dates between 1 A.D. and 9999 A.D in the Gregorian calendar system.

Julian Day, Modified Julian Day, and the Gregorian Calendar

PowerCenter supports dates in the Gregorian calendar system only. Dates expressed in a different calendar system are not supported. Dates in the Julian calendar are called Julian *dates* and are not supported in PowerCenter. This term should not be confused with Julian *Day* or with Modified Julian Day.

PowerCenter provides the ability to manipulate Modified Julian Day (MJD) formats using the J format string. The MJD for a given date is the number of days to that date since Jan 1 4713 B.C. 00:00:00 (midnight). By definition, MJD includes a time component expressed as a decimal, which represents some fraction of 24 hours. The J format string does not convert this time component.

For example, the following TO_DATE expression converts strings in the SHIP_DATE_MJD_STRING port to date values in the default date format:

```
TO_DATE (SHIP_DATE_MJD_STR, 'J')
```

SHIP_DATE_MJD_STR	RETURN_VALUE
2451544	Dec 31 1999 00:00:00.000000000
2415021	Jan 1 1900 00:00:00.000000000

Because the J format string does not include the time portion of a date, the return values have the time set to 00:00:00.000000000.

You can also use the J format string in TO_CHAR expressions. For example, use the J format string in a TO_CHAR expression to convert date values to MJD values expressed as strings. For example:

```
TO_CHAR(SHIP_DATE, 'J')
```

SHIP_DATE	RETURN_VALUE
Dec 31 1999 23:59:59	2451544
Jan 1 1900 01:02:03	2415021

Note: The Integration Service ignores the time portion of the date in a TO_CHAR expression.

Dates in the Year 2000

All transformation language date functions support the year 2000. PowerCenter supports dates between 1 A.D. and 9999 A.D.

RR Format String

The transformation language provides the RR format string to convert strings with two-digit years to dates. Using TO_DATE and the RR format string, you can convert a string in the format MM/DD/RR to a date. The RR format string converts data differently depending on the current year.

- **Current Year Between 0 and 49.** If the current year is between 0 and 49 (such as 2003) and the source string year is between 0 and 49, the Integration Service returns the current century plus the two-digit year from the source string. If the source string year is between 50 and 99, the Integration Service returns the previous century plus the two-digit year from the source string.
- **Current Year Between 50 and 99.** If the current year is between 50 and 99 (such as 1998) and the source string year is between 0 and 49, the Integration Service returns the next century plus the two-digit year from the source string. If the source string year is between 50 and 99, the Integration Service returns the current century plus the specified two-digit year.

The following table summarizes how the RR format string converts to dates:

Current year	Source year	RR Format String Returns
0-49	0-49	Current century
0-49	50-99	Previous century
50-99	0-49	Next century
50-99	50-99	Current century

Example

The following expression produces the same return values for any current year between 1950 and 2049:

```
TO_DATE( ORDER_DATE, 'MM/DD/RR' )

ORDER_DATE              RETURN_VALUE
'04/12/98'              04/12/1998 00:00:00.000000000
'11/09/01'              11/09/2001 00:00:00.000000000
```

Difference Between the YY and RR Format Strings

PowerCenter also provides a YY format string. Both the RR and YY format strings specify two-digit years. The YY and RR format strings produce identical results when used with all date functions except TO_DATE. In TO_DATE expressions, RR and YY produce different results.

The following table shows the different results each format string returns:

String	Current Year	TO_DATE(String, 'MM/DD/RR')	TO_DATE(String, 'MM/DD/YY')
04/12/98	1998	04/12/1998 00:00:00.000000000	04/12/1998 00:00:00.000000000
11/09/01	1998	11/09/2001 00:00:00.000000000	11/09/1901 00:00:00.000000000
04/12/98	2003	04/12/1998 00:00:00.000000000	04/12/2098 00:00:00.000000000
11/09/01	2003	11/09/2001 00:00:00.000000000	11/09/2001 00:00:00.000000000

For dates in the year 2000 and beyond, the YY format string produces less meaningful results than the RR format string. Use the RR format string for dates in the twenty-first century.

Dates in Relational Databases

In general, dates stored in relational databases contain a date and time value. The date includes the month, day, and year, while the time might include the hours, minutes, seconds, and sub-seconds. You can pass datetime data to any of the date functions.

Dates in Flat Files

Use the TO_DATE function to convert strings to datetime values. You can also use IS_DATE to check if a string is a valid date before converting it with TO_DATE. The transformation language date functions accept date values only. To pass a string to a date function, you must first use the TO_DATE function to convert it to a transformation Date/Time datatype.

Default Date Format

The Integration Service uses a default date format to store and manipulate strings that represent dates. To specify the default date format, enter a date format in the DateTime Format String attribute on the Config Object tab for a session or session configuration object. By default, the date format is MM/DD/YYYY HH24:MI:SS.US.

Because PowerCenter stores dates in binary format, the Integration Service uses the default date format when you perform the following actions:

- **Convert a date to a string by connecting a date/time port to a string port.** The Integration Service converts the date to a string in the date format defined in the session configuration object.
- **Convert a string to a date by connecting a string port to a date/time port.** The Integration Service expects the string values to be in the date format defined by the session configuration object. If an input value does not match this format, or if it is an invalid date, the Integration Service skips the row. If the string is in this format, the Integration Service converts the string to a date value.
- **Use TO_CHAR(date, [format_string]) to convert dates to strings.** If you omit the format string, the Integration Service returns the string in the date format defined in the session properties. If you specify a format string, the Integration Service returns a string in the specified format.
- **Use TO_DATE(date, [format_string]) to convert strings to dates.** If you omit the format string, the Integration Service expects the string in the date format defined in the session properties. If you specify a format string, the Integration Service expects a string in the specified format.

The default date format of MM/DD/YYYY HH24:MI:SS.US consists of:

- Month (January = 01, September = 09)
- Day (of the month)
- Year (expressed in four digits, such as 1998)
- Hour (in 24-hour format, for example, 12:00:00AM = 0, 1:00:00AM = 1, 12:00:00PM = 12, 11:00:00PM = 23)
- Minutes
- Seconds
- Microseconds

Date Format Strings

You can evaluate input dates using a combination of format strings and date functions. Date format strings are not internationalized and must be entered in predefined formats as listed in the following table.

The following table summarizes the format strings to specify a part of a date:

Format String	Description
D, DD, DDD, DAY, DY, J	Days (01-31). Use any of these format strings to specify the entire day portion of a date. For example, if you pass 12-APR-1997 to a date function, use any of these format strings specify 12.
HH, HH12, HH24	Hour of day (0-23), where 0 is 12 AM (midnight). Use any of these formats to specify the entire hour portion of a date. For example, if you pass the date 12-APR-1997 2:01:32 PM, use HH, HH12, or HH24 to specify the hour portion of the date.
MI	Minutes (0-59).

Format String	Description
MM, MON, MONTH	Month (01-12). Use any of these format strings to specify the entire month portion of a date. For example, if you pass 12-APR-1997 to a date function, use MM, MON, or MONTH to specify APR.
MS	Milliseconds (0-999).
NS	Nanoseconds (0-999999999).
SS, SSSS	Seconds (0-59).
US	Microseconds (0-999999).
Y, YY, YYY, YYYY, RR	Year portion of date (0001 to 9999). Use any of these format strings to specify the entire year portion of a date. For example, if you pass 12-APR-1997 to a date function, use Y, YY, YYY, or YYYY to specify 1997.

Note: The format string is not case sensitive. It must always be enclosed within single quotation marks.

The following table describes date functions that use date format strings to evaluate input dates:

Function	Description
ADD_TO_DATE	The part of the date you want to change.
DATE_DIFF	The part of the date to use to calculate the difference between two dates.
GET_DATE_PART	The part of the date you want to return. This function returns an integer value based on the default date format.
IS_DATE	The date you want to check.
ROUND	The part of the date you want to round.
SET_DATE_PART	The part of the date you want to change.
SYSTIMESTAMP	The timestamp precision.
TO_CHAR (Dates)	The character string.
TO_DATE	The character string.
TRUNC (Dates)	The part of the date you want to truncate.

TO_CHAR Format Strings

The TO_CHAR function converts a Date/Time datatype to a string with the format you specify. You can convert the entire date or a part of the date to a string. You might use TO_CHAR to convert dates to strings, changing the format for reporting purposes.

TO_CHAR is generally used when the target is a flat file or a database that does not support a Date/Time datatype.

The following table summarizes the format strings for dates in the function TO_CHAR:

Format String	Description
AM, A.M., PM, P.M.	Meridian indicator. Use any of these format strings to specify AM and PM hours. AM and PM return the same values as A.M. and P.M.
D	Day of week (1-7), where Sunday equals 1.
DAY	Name of day, including up to nine characters (for example, Wednesday).
DD	Day of month (01-31).
DDD	Day of year (001-366, including leap years).
DY	Abbreviated three-character name for a day (for example, Wed).
HH, HH12	Hour of day (01-12).
HH24	Hour of day (00-23), where 00 is 12AM (midnight).
J	Modified Julian Day. Converts the calendar date to a string equivalent to its Modified Julian Day value, calculated from Jan 1, 4713 00:00:00 B.C. It ignores the time component of the date. For example, the expression TO_CHAR(SHIP_DATE, 'J') converts Dec 31 1999 23:59:59 to the string 2451544.
MI	Minutes (00-59).
MM	Month (01-12).
MONTH	Name of month, including up to nine characters (for example, January).
MON	Abbreviated three-character name for a month (for example, Jan).
MS	Milliseconds (0-999).
NS	Nanoseconds (0-999999999).
Q	Quarter of year (1-4), where January to March equals 1.
RR	Last two digits of a year. The function removes the leading digits. For example, if you use 'RR' and pass the year 1997, TO_CHAR returns 97. When used with TO_CHAR, 'RR' produces the same results as, and is interchangeable with, 'YY.' However, when used with TO_DATE, 'RR' calculates the closest appropriate century and supplies the first two digits of the year.
SS	Seconds (00-59).
SSSSS	Seconds since midnight (00000 - 86399). When you use SSSSS in a TO_CHAR expression, the Integration Service only evaluates the time portion of a date. For example, the expression TO_CHAR(SHIP_DATE, 'MM/DD/YYYY SSSSS') converts 12/31/1999 01:02:03 to 12/31/1999 03723.
US	Microseconds (0-999999).
Y	Last digit of a year. The function removes the leading digits. For example, if you use 'Y' and pass the year 1997, TO_CHAR returns 7.
YY	Last two digits of a year. The function removes the leading digits. For example, if you use 'YY' and pass the year 1997, TO_CHAR returns 97.

Format String	Description
YYY	Last three digits of a year. The function removes the leading digits. For example, if you use 'YYY' and pass the year 1997, TO_CHAR returns 997.
YYYY	Entire year portion of date. For example, if you use 'YYYY' and pass the year 1997, TO_CHAR returns 1997.
W	Week of month (1-5), where week 1 starts on the first day of the month and ends on the seventh, week 2 starts on the eighth day and ends on the fourteenth day. For example, Feb 1 designates the first week of February.
WW	Week of year (01-53), where week 01 starts on Jan 1 and ends on Jan 7, week 2 starts on Jan 8 and ends on Jan 14, and so on.
- / . ; :	Punctuation that displays in the output. You might use these symbols to separate date parts. For example, you create the following expression to separate date parts with a period: TO_CHAR(DATES, 'MM.DD.YYYY').
"text"	Text that displays in the output. For example, if you create an output port with the expression: TO_CHAR(DATES, 'MM/DD/YYYY "Sales Were Up"') and pass the date Apr 1 1997, the function returns the string '04/01/1997 Sales Were Up'. You can enter multibyte characters that are valid in the repository code page.
""	Use double quotation marks to separate ambiguous format strings, for example D""DDD. The empty quotation marks do not appear in the output.

Note: The format string is not case sensitive. It must always be enclosed within single quotation marks.

Examples

The following examples illustrate the J, SSSSS, RR, and YY format strings. See the individual functions for more examples.

Note: The Integration Service ignores the time portion of the date in a TO_CHAR expression.

J Format String

Use the J format string in a TO_CHAR expression to convert date values to MJD values expressed as strings. For example:

```
TO_CHAR(SHIP_DATE, 'J')
```

SHIP_DATE	RETURN_VALUE
Dec 31 1999 23:59:59	2451544
Jan 1 1900 01:02:03	2415021

SSSSS Format String

You can also use the format string SSSSS in a TO_CHAR expression. For example, the following expression converts the dates in the SHIP_DATE port to strings representing the total seconds since midnight:

```
TO_CHAR( SHIP_DATE, 'SSSSS')
```

SHIP_DATE	RETURN_VALUE
12/31/1999 01:02:03	3723
09/15/1996 23:59:59	86399

RR Format String

The following expression converts dates to strings in the format MM/DD/YY:

```
TO_CHAR( SHIP_DATE, 'MM/DD/RR')
```

SHIP_DATE	RETURN_VALUE
12/31/1999 01:02:03	12/31/99
09/15/1996 23:59:59	09/15/96
05/17/2003 12:13:14	05/17/03

YY Format String

In TO_CHAR expressions, the YY format string produces the same results as the RR format string. The following expression converts dates to strings in the format MM/DD/YY:

```
TO_CHAR( SHIP_DATE, 'MM/DD/YY')
```

SHIP_DATE	RETURN_VALUE
12/31/1999 01:02:03	12/31/99
09/15/1996 23:59:59	09/15/96
05/17/2003 12:13:14	05/17/03

TO_DATE and IS_DATE Format Strings

The TO_DATE function converts a string with the format you specify to a datetime value. TO_DATE is generally used to convert strings from flat files to datetime values. TO_DATE format strings are not internationalized and must be entered in predefined formats as listed in “TO_DATE and IS_DATE Format Strings” on page 30.

Note: TO_DATE and IS_DATE use the same set of format strings.

When you create a TO_DATE expression, use a format string for each part of the date in the source string. The source string format and the format string must match, including any date separator. If any part does not match, the Integration Service does not convert the string, and it skips the row. If you omit the format string, the source string must be in the date format specified in the session.

IS_DATE indicates if a value is a valid date. A valid date is any string representing a valid date in the date format specified in the session. If the strings you want to test are not in this date format, use the format strings listed in “TO_DATE and IS_DATE Format Strings” on page 30 to specify the date format. If a string does not match the specified format string or is not a valid date, the function returns FALSE (0). If the string matches the format string and is a valid date, the function returns TRUE (1). IS_DATE format strings are not internationalized and must be entered in predefined formats as listed in the following table.

The following table summarizes the format strings for the functions TO_DATE and IS_DATE:

Table 1. TO_DATE and IS_DATE Format Strings

Format String	Description
AM, a.m., PM, p.m.	Meridian indicator. Use any of these format strings to specify AM and PM hours. AM and PM return the same values as do a.m. and p.m.
DAY	Name of day, including up to nine characters (for example, Wednesday). The DAY format string is not case sensitive.
DD	Day of month (1-31).

Format String	Description
DDD	Day of year (001-366, including leap years).
DY	Abbreviated three-character name for a day (for example, Wed). The DY format string is not case sensitive.
HH, HH12	Hour of day (1-12).
HH24	Hour of day (0-23), where 0 is 12AM (midnight).
J	Modified Julian Day. Convert strings in MJD format to date values. It ignores the time component of the source string, assigning all dates the time of 00:00:00.000000000. For example, the expression TO_DATE('2451544', 'J') converts 2451544 to Dec 31 1999 00:00:00. 000000000.
MI	Minutes (0-59).
MM	Month (1-12).
MONTH	Name of month, including up to nine characters (for example, August). Case does not matter.
MON	Abbreviated three-character name for a month (for example, Aug). Case does not matter.
MS	Milliseconds (0-999).
NS	Nanoseconds (0-999999999).
RR	Four-digit year (for example, 1998, 2034). Use when source strings include two-digit years. Use with TO_DATE to convert two-digit years to four-digit years. - Current Year Between 50 and 99. If the current year is between 50 and 99 (such as 1998) and the year value of the source string is between 0 and 49, the Integration Service returns the next century plus the two-digit year from the source string. If the year value of the source string is between 50 and 99, the Integration Service returns the current century plus the specified two-digit year. - Current Year Between 0 and 49. If the current year is between 0 and 49 (such as 2003) and the source string year is between 0 and 49, the Integration Service returns the current century plus the two-digit year from the source string. If the source string year is between 50 and 99, the Integration Service returns the previous century plus the two-digit year from the source string.
SS	Seconds (0-59).
SSSSS	Seconds since midnight. When you use SSSSS in a TO_DATE expression, the Integration Service only evaluates the time portion of a date. For example, the expression TO_DATE(DATE_STR, 'MM/DD/YYYY SSSSS') converts 12/31/1999 3783 to 12/31/1999 01:02:03.
US	Microseconds (0-999999).
Y	The current year on the node running the Integration Service with the last digit of the year replaced with the string value.
YY	The current year on the node running the Integration Service with the last two digits of the year replaced with the string value.

Format String	Description
YYY	The current year on the node running the Integration Service with the last three digits of the year replaced with the string value.
YYYY	Four digits of a year. Do not use this format string if you are passing two-digit years. Use the RR or YY format string instead.

Rules and Guidelines for Date Format Strings

Use the following rules and guidelines when you work with date format strings:

- The format of the TO_DATE string must match the format string including any date separators. If it does not, the Integration Service might return inaccurate values or skip the row. For example, if you pass the string '20200512', representing May 12, 2020, to TO_DATE, you must include the format string YYYYMMDD. If you do not include a format string, the Integration Service expects the string in the date format specified in the session. Likewise, if you pass a string that does not match the format string, the Integration Service returns an error and skips the row. For example, if you pass the string 2020120 to TO_DATE and include the format string YYYYMMDD, the Integration Service returns an error and skips the row because the string does not match the format string.
- The format string must be enclosed within single quotation marks.
- The Integration Service uses the default date time format specified in the session. Default is MM/DD/YYYY HH24:MI:SS.US. The format string is not case sensitive.

Example

The following examples illustrate the J, RR, and SSSSS format strings. See the individual functions for more examples.

J Format String

The following expression converts strings in the SHIP_DATE_MJD_STRING port to date values in the default date format:

```
TO_DATE (SHIP_DATE_MJD_STR, 'J')
```

```
SHIP_DATE_MJD_STR                RETURN_VALUE
2451544                          Dec 31 1999 00:00:00.000000000
2415021                          Jan 1 1900 00:00:00.000000000
```

Because the J format string does not include the time portion of a date, the return values have the time set to 00:00:00.000000000.

RR Format String

The following expression converts a string to a four-digit year format. The current year is 1998:

```
TO_DATE( DATE_STR, 'MM/DD/RR')
```

```
DATE_STR            RETURN VALUE
04/01/98            04/01/1998 00:00:00.000000000
08/17/05            08/17/2005 00:00:00.000000000
```

YY Format String

The following expression converts a string to a four-digit year format. The current year is 1998:

```
TO_DATE( DATE_STR, 'MM/DD/YY')
```

DATE_STR	RETURN VALUE
04/01/98	04/01/1998 00:00:00.000000000
08/17/05	08/17/1905 00:00:00.000000000

Note: For the second row, RR returns the year 2005, but YY returns the year 1905.

SSSSS Format String

The following expression converts strings that include the seconds since midnight to date values:

```
TO_DATE( DATE_STR, 'MM/DD/YYYY SSSSS')
```

DATE_STR	RETURN_VALUE
12/31/1999 3783	12/31/1999 01:02:03.000000000
09/15/1996 86399	09/15/1996 23:59:59.000000000

Understanding Date Arithmetic

The transformation language provides built-in date functions so you can perform arithmetic on datetime values as follows:

- **ADD_TO_DATE.** Add or subtract a specific portion of a date.
- **DATE_DIFF.** Subtract two dates.
- **SET_DATE_PART.** Change one part of a date.

You cannot use numeric arithmetic operators (such as + or -) to add or subtract dates.

The transformation language recognizes leap years and accepts dates between Jan. 1, 0001 00:00:00.000000000 A.D. and Dec. 31, 9999 23:59:59.999999999 A.D.

Note: The transformation language uses the transformation Date/Time datatype to specify date values. You can only use the date functions on datetime values.

CHAPTER 6

Functions

This chapter describes the functions in the transformation language in alphabetical order. Each function description includes:

- Syntax
- Return value
- Example

Function Categories

The transformation language provides the following types of functions:

- Aggregate
- Character
- Conversion
- Data Cleansing
- Date
- Encoding
- Financial
- Numerical
- Scientific
- Special
- String
- Test
- Variable

Aggregate Functions

Aggregate functions return summary values for non-null values in selected ports. With aggregate functions you can:

- Calculate a single value for all rows in a group.
- Return a single value for each group in an Aggregator transformation.
- Apply filters to calculate values for specific rows in the selected ports.

- Use operators to perform arithmetic within the function.
- Calculate two or more aggregate values derived from the same source columns in a single pass.

The transformation language includes the following aggregate functions:

- AVG
- COUNT
- FIRST
- LAST
- MAX (Date)
- MAX (Number)
- MAX (String)
- MEDIAN
- MIN (Date)
- MIN (Number)
- MIN (String)
- PERCENTILE
- STDDEV
- SUM
- VARIANCE

If you configure the Integration Service to run in Unicode mode, MIN and MAX return values according to the sort order of the code page you specify in the session properties.

Use aggregate functions in Aggregator transformations only. You can nest only one aggregate function within another aggregate function. The Integration Service evaluates the innermost aggregate function expression and uses the result to evaluate the outer aggregate function expression. You can set up an Aggregator transformation that groups by ID and nests two aggregate functions as follows:

```
SUM( AVG( earnings ) )
```

where the dataset contains the following values:

ID	EARNINGS
1	32
1	45
1	100
2	65
2	75
2	76
3	21
3	45
3	99

The return value is 186. The Integration Service groups by ID, evaluates the AVG expression, and returns three values. Then it adds the values with the SUM function to get the result.

Aggregate Functions and Nulls

When you configure the Integration Service, you can choose how you want to handle null values in aggregate functions. You can have the Integration Service treat null values in aggregate functions as NULL or 0.

By default, the Integration Service treats null values as NULL in aggregate functions. If you pass an entire port of null values, the function returns NULL. You can optionally configure the Integration Service if you pass an entire port of null values to an aggregate function to return 0.

Filter Conditions

Use a filter condition to limit the rows returned in a search.

A filter limits the rows returned in a search. You can apply a filter condition to all aggregate functions and to CUME, MOVINGAVG, and MOVINGSUM. The filter condition must evaluate to TRUE, FALSE, or NULL. If the filter condition evaluates to NULL or FALSE, the Integration Service does not select the row.

You can enter any valid transformation expression. For example, the following expression calculates the median salary for all employees who make more than $50,000:

```
MEDIAN( SALARY, SALARY > 50000 )
```

You can also use other numeric values as the filter condition. For example, you can enter the following as the complete syntax for the MEDIAN function, including a numeric port:

```
MEDIAN( PRICE, QUANTITY > 0 )
```

In all cases, the Integration Service rounds a decimal value to an integer (for example, 1.5 to 2, 1.2 to 1, 0.35 to 0) for the filter condition. If the value rounds to 0, the filter condition returns FALSE. If you do not want to round up a value, use the TRUNC function to truncate the value to an integer:

```
MEDIAN( PRICE, TRUNC( QUANTITY ) > 0 )
```

If you omit the filter condition, the function selects all rows in the port.

Character Functions

The transformation language includes the following character functions:

- ASCII
- CHR
- CHRCODE
- CONCAT
- INITCAP
- INSTR
- LENGTH
- LOWER
- LPAD
- LTRIM
- METAPHONE
- REPLACECHR
- REPLACESTR
- RPAD
- RTRIM
- SOUNDEX
- SUBSTR
- UPPER

The character functions MAX, MIN, LOWER, UPPER, and INITCAP use the code page of the Integration Service to evaluate character data.

Conversion Functions

The transformation language includes the following conversion functions:

- TO_BIGINT
- TO_CHAR(Number)
- TO_DATE
- TO_DECIMAL
- TO_FLOAT
- TO_INTEGER

Data Cleansing Functions

The transformation language includes a group of functions to eliminate data errors. You can complete the following tasks with data cleansing functions:

- Test input values.
- Convert the datatype of an input value.
- Trim string values.
- Replace characters in a string.
- Encode strings.
- Match patterns in regular expressions.

The transformation language includes the following data cleansing functions:

- GREATEST
- IN
- INSTR
- IS_DATE
- IS_NUMBER
- IS_SPACES
- ISNULL
- LEAST
- LTRIM
- METAPHONE
- REG_EXTRACT
- REG_MATCH
- REG_REPLACE
- REPLACECHR
- REPLACESTR
- RTRIM
- SOUNDEX
- SUBSTR

- TO_BIGINT
- TO_CHAR
- TO_DATE
- TO_DECIMAL
- TO_FLOAT
- TO_INTEGER

Date Functions

The transformation language includes a group of date functions to round, truncate, or compare dates, extract one part of a date, or perform arithmetic on a date.

You can pass any value with a date datatype to any of the date functions. However, if you want to pass a string to a date function, you must first use the TO_DATE function to convert it to a transformation Date/Time datatype.

The transformation language includes the following date functions:

- ADD_TO_DATE
- DATE_COMPARE
- DATE_DIFF
- GET_DATE_PART
- IS_DATE
- LAST_DAY
- MAKE_DATE_TIME
- MAX
- MIN
- ROUND(Date)
- SET_DATE_PART
- SYSTIMESTAMP
- TO_CHAR(Date)
- TRUNC(Date)

Several of the date functions include a *format* argument. You must specify one of the transformation language format strings for this argument. Date format strings are not internationalized.

The Date/Time transformation datatype supports dates with precision to the nanosecond.

Related Topics:

- “Date Format Strings” on page 26

Encoding Functions

The transformation language includes the following functions for data encryption, compression, encoding, and checksum:

- AES_DECRYPT
- AES_ENCRYPT
- COMPRESS

- CRC32
- DEC_BASE64
- DECOMPRESS
- ENC_BASE64
- MD5

Financial Functions

The transformation language includes the following financial functions:

- FV
- NPER
- PMT
- PV
- RATE

Numeric Functions

The transformation language includes the following numeric functions:

- ABS
- CEIL
- CONV
- CUME
- EXP
- FLOOR
- LN
- LOG
- MAX
- MIN
- MOD
- MOVINGAVG
- MOVINGSUM
- POWER
- RAND
- ROUND
- SIGN
- SQRT
- TRUNC

Scientific Functions

The transformation language includes the following scientific functions:

- COS
- COSH
- SIN
- SINH
- TAN
- TANH

Special Functions

The transformation language includes the following special functions:

- ABORT
- DECODE
- ERROR
- IIF
- LOOKUP

Generally, you use special functions in Expression, Filter, and Update Strategy transformations. You can nest other functions within special functions. You can also nest a special function in an aggregate function.

String Functions

The transformation language includes the following string functions:

- CHOOSE
- INDEXOF
- MAX
- MIN
- REVERSE

Test Functions

The transformation language includes the following test functions:

- ISNULL
- IS_DATE
- IS_NUMBER
- IS_SPACES

Variable Functions

The transformation language includes a group of variable functions to update the current value of a mapping variable throughout the session. When you run a workflow, the Integration Service evaluates the start and current

value of a variable at the beginning of the session based on the final value of the variable from the last session run. Use the following variable functions:

- SetCountVariable
- SetMaxVariable
- SetMinVariable
- SetVariable

Use different variable functions with a variable based on the aggregation type of the variable.

When using mapping variables in sessions with multiple partitions, use variable functions to determine the final value of the variable for each partition. At the end of the session, the Integration Service performs the aggregate function across all partitions to determine one final value to save to the repository. Unless overridden, it uses the saved value as the start value of the variable for the next time you use this session.

For example, you use SetMinVariable to set a variable to the minimum evaluated value. The Integration Service calculates the minimum current value for the variable for each partition. Then at the end of the session, it finds the minimum current value across all partitions and saves that value into the repository.

Use SetVariable only once for each mapping variable in a pipeline. When you create multiple partitions in a pipeline, the Integration Service uses multiple threads to process that pipeline. If you use this function more than once for the same variable, the current value of a mapping variable may have indeterministic results.

ABORT

Availability:
Designer

Stops the session, and issues a specified error message to the session log file. When the Integration Service encounters an ABORT function, it stops transforming data at that row. It processes any rows read before the session aborts and loads them based on the source- or target-based commit interval and the buffer block size defined for the session. The Integration Service writes to the target up to the aborted row and then rolls back all uncommitted data to the last commit point. You can perform recovery on the session after rollback.

Use ABORT to validate data. Generally, you use ABORT within an IIF or DECODE function to set rules for aborting a session.

Use the ABORT function for both input and output port default values. You might use ABORT for input ports to keep null values from passing into a transformation. You can also use ABORT to handle any kind of transformation error, including ERROR function calls within an expression. The default value overrides the ERROR function in an expression. If you want to ensure the session stops when an error occurs, assign ABORT as the default value.

If you use ABORT in an expression for an unconnected port, the Integration Service does not run the ABORT function.

Note: The Integration Service handles the ABORT function and the Abort command you issue from the Workflow Manager differently.

Syntax

```
ABORT( string )
```

Argument	Required/ Optional	Description
string	Required	String. The message you want to display in the session log file when the session stops. The string can be any length. You can enter any valid transformation expression.

Return Value

NULL.

ABS

Availability:
Designer
Workflow Manager

Returns the absolute value of a numeric value.

Syntax

```
ABS( numeric_value )
```

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. Passes the values for which you want to return the absolute values. You can enter any valid transformation expression.

Return Value

Positive numeric value. ABS returns the same datatype as the numeric value passed as an argument. If you pass a Double, it returns a Double. Likewise, if you pass an Integer, it returns an Integer.

NULL if you pass a null value to the function.

Note: If the return value is Decimal with precision greater than 15, you can enable high precision in the session properties to ensure decimal precision up to 28 digits.

Example

The following expression returns the difference between two numbers as a positive value, regardless of which number is larger:

```
ABS( PRICE - COST )
```

PRICE	COST	RETURN VALUE
250	150	100
52	48	4
169.95	69.95	100
59.95	NULL	NULL
70	30	40

PRICE	COST	RETURN VALUE
430	330	100
100	200	100

ADD_TO_DATE

Availability:
Designer
Workflow Manager

Adds a specified amount to one part of a datetime value, and returns a date in the same format as the date you pass to the function. ADD_TO_DATE accepts positive and negative integer values. Use ADD_TO_DATE to change the following parts of a date:

- **Year.** Enter a positive or negative integer in the *amount* argument. Use any of the year format strings: Y, YY, YYY, or YYYY. The following expression adds 10 years to all dates in the SHIP_DATE port:

  ```
  ADD_TO_DATE ( SHIP_DATE, 'YY', 10 )
  ```

- **Month.** Enter a positive or negative integer in the *amount* argument. Use any of the month format strings: MM, MON, MONTH. The following expression subtracts 10 months from each date in the SHIP_DATE port:

  ```
  ADD_TO_DATE( SHIP_DATE, 'MONTH', -10 )
  ```

- **Day.** Enter a positive or negative integer in the *amount* argument. Use any of the day format strings: D, DD, DDD, DY, and DAY. The following expression adds 10 days to each date in the SHIP_DATE port:

  ```
  ADD_TO_DATE( SHIP_DATE, 'DD', 10 )
  ```

- **Hour.** Enter a positive or negative integer in the *amount* argument. Use any of the hour format strings: HH, HH12, HH24. The following expression adds 14 hours to each date in the SHIP_DATE port:

  ```
  ADD_TO_DATE( SHIP_DATE, 'HH', 14 )
  ```

- **Minute.** Enter a positive or negative integer in the *amount* argument. Use the MI format string to set the minute. The following expression adds 25 minutes to each date in the SHIP_DATE port:

  ```
  ADD_TO_DATE( SHIP_DATE, 'MI', 25 )
  ```

- **Seconds.** Enter a positive or negative integer in the *amount* argument. Use the SS format string to set the second. The following expression adds 59 seconds to each date in the SHIP_DATE port:

  ```
  ADD_TO_DATE( SHIP_DATE, 'SS', 59 )
  ```

- **Milliseconds.** Enter a positive or negative integer in the *amount* argument. Use the MS format string to set the milliseconds. The following expression adds 125 milliseconds to each date in the SHIP_DATE port:

  ```
  ADD_TO_DATE( SHIP_DATE, 'MS', 125 )
  ```

- **Microseconds.** Enter a positive or negative integer in the *amount* argument. Use the US format string to set the microseconds. The following expression adds 2,000 microseconds to each date in the SHIP_DATE port:

  ```
  ADD_TO_DATE( SHIP_DATE, 'US', 2000 )
  ```

- **Nanoseconds.** Enter a positive or negative integer in the *amount* argument. Use the NS format string to set the nanoseconds. The following expression adds 3,000,000 nanoseconds to each date in the SHIP_DATE port:

  ```
  ADD_TO_DATE( SHIP_DATE, 'NS', 3000000 )
  ```

Syntax

```
ADD_TO_DATE( date, format, amount )
```

Argument	Required/ Optional	Description
date	Required	Date/Time datatype. Passes the values you want to change. You can enter any valid transformation expression.
format	Required	A format string specifying the portion of the date value you want to change. Enclose the format string within single quotation marks, for example, 'mm'. The format string is not case sensitive.
amount	Required	An integer value specifying the amount of years, months, days, hours, and so on by which you want to change the date value. You can enter any valid transformation expression that evaluates to an integer.

Return Value

Date in the same format as the date you pass to this function.

NULL if a null value is passed as an argument to the function.

Examples

The following expressions all add one month to each date in the DATE_SHIPPED port. If you pass a value that creates a day that does not exist in a particular month, the Integration Service returns the last day of the month. For example, if you add one month to Jan 31 1998, the Integration Service returns Feb 28 1998.

Also note, ADD_TO_DATE recognizes leap years and adds one month to Jan 29 2000:

```
ADD_TO_DATE( DATE_SHIPPED, 'MM', 1 )
ADD_TO_DATE( DATE_SHIPPED, 'MON', 1 )
ADD_TO_DATE( DATE_SHIPPED, 'MONTH', 1 )
```

DATE_SHIPPED	RETURN VALUE	
Jan 12 1998 12:00:30AM	Feb 12 1998 12:00:30AM	
Jan 31 1998 6:24:45PM	Feb 28 1998 6:24:45PM	
Jan 29 2000 5:32:12AM	Feb 29 2000 5:32:12AM	*(Leap Year)*
Oct 9 1998 2:30:12PM	Nov 9 1998 2:30:12PM	
NULL	NULL	

The following expressions subtract 10 days from each date in the DATE_SHIPPED port:

```
ADD_TO_DATE( DATE_SHIPPED, 'D', -10 )
ADD_TO_DATE( DATE_SHIPPED, 'DD', -10 )
ADD_TO_DATE( DATE_SHIPPED, 'DDD', -10 )
ADD_TO_DATE( DATE_SHIPPED, 'DY', -10 )
ADD_TO_DATE( DATE_SHIPPED, 'DAY', -10 )
```

DATE_SHIPPED	RETURN VALUE	
Jan 1 1997 12:00:30AM	Dec 22 1996 12:00AM	
Jan 31 1997 6:24:45PM	Jan 21 1997 6:24:45PM	
Mar 9 1996 5:32:12AM	Feb 29 1996 5:32:12AM	*(Leap Year)*
Oct 9 1997 2:30:12PM	Sep 30 1997 2:30:12PM	
Mar 3 1996 5:12:20AM	Feb 22 1996 5:12:20AM	
NULL	NULL	

The following expressions subtract 15 hours from each date in the DATE_SHIPPED port:

```
ADD_TO_DATE( DATE_SHIPPED, 'HH', -15 )
ADD_TO_DATE( DATE_SHIPPED, 'HH12', -15 )
ADD_TO_DATE( DATE_SHIPPED, 'HH24', -15 )
```

DATE_SHIPPED	RETURN VALUE	
Jan 1 1997 12:00:30AM	Dec 31 1996 9:00:30AM	
Jan 31 1997 6:24:45PM	Jan 31 1997 3:24:45AM	
Oct 9 1997 2:30:12PM	Oct 8 1997 11:30:12PM	
Mar 3 1996 5:12:20AM	Mar 2 1996 2:12:20PM	
Mar 1 1996 5:32:12AM	Feb 29 1996 2:32:12PM	*(Leap Year)*
NULL	NULL	

Working with Dates

Use the following tips when working with ADD_TO_DATE:

- You can add or subtract any part of the date by specifying a format string and making the *amount* argument a positive or negative integer.
- If you pass a value that creates a day that does not exist in a particular month, the Integration Service returns the last day of the month. For example, if you add one month to Jan 31 1998, the Integration Service returns Feb 28 1998.
- You can nest TRUNC and ROUND to manipulate dates.
- You can nest TO_DATE to convert strings to dates.
- ADD_TO_DATE changes only one portion of the date, which you specify. If you modify a date so that it changes from standard to daylight savings time, you need to change the hour portion of the date.

AES_DECRYPT

Availability:
Designer
Workflow Manager

Returns decrypted data to string format. The Integration Service uses Advanced Encryption Standard (AES) algorithm with 128-bit encoding. The AES algorithm is a FIPS-approved cryptographic algorithm.

Syntax

```
AES_DECRYPT ( value, key )
```

Argument	Required/ Optional	Description
value	Required	Binary datatype. Value you want to decrypt.
key	Required	String datatype. Precision of 16 characters or fewer. You can use mapping variables for the key. Use the same key to decrypt a value that you used to encrypt it.

Return Value

Decrypted binary value.

NULL if the input value is a null value.

Example

The following example returns decrypted social security numbers. In this example, the Integration Service derives the key from the first three numbers of the social security number using the SUBSRT function:

```
AES_DECRYPT( SSN_ENCRYPT, SUBSTR( SSN,1,3 ))
```

```
SSN_ENCRYPT                                DECRYPTED VALUE
07FB945926849D2B1641E708C85E4390           832-17-1672
9153ACAB89D65A4B81AD2ABF151B099D           832-92-4731
AF6B5E4E39F974B3F3FB0F22320CC60B           832-46-7552
992D6A5D91E7F59D03B940A4B1CBBCBE           832-53-6194
992D6A5D91E7F59D03B940A4B1CBBCBE           832-81-9528
```

AES_ENCRYPT

Availability:
Designer
Workflow Manager

Returns data in encrypted format. The Integration Service uses Advanced Encryption Standard (AES) algorithm with 128-bit encoding. The AES algorithm is a FIPS-approved cryptographic algorithm.

Use this function to prevent sensitive data from being visible to everyone. For example, to store social security numbers in a data warehouse, use the AES_ENCRYPT function to encrypt the social security numbers to maintain confidentiality.

Syntax

```
AES_ENCRYPT ( value, key )
```

Argument	Required/ Optional	Description
value	Required	String datatype. Value you want to encrypt.
key	Required	String datatype. Precision of 16 characters or fewer. You can use mapping variables for the key.

Return Value

Encrypted binary value.

NULL if the input is a null value.

Example

The following example returns encrypted values for social security numbers. In this example, the Integration Service derives the key from the first three numbers of the social security number using the SUBSTR function:

```
AES_ENCRYPT( SSN, SUBSTR( SSN,1,3 ))
```

```
SSN                 ENCRYPTED VALUE
832-17-1672         07FB945926849D2B1641E708C85E4390
832-92-4731         9153ACAB89D65A4B81AD2ABF151B099D
832-46-7552         AF6B5E4E39F974B3F3FB0F22320CC60B
```

```
SSN                  ENCRYPTED VALUE
832-53-6194          992D6A5D91E7F59D03B940A4B1CBBCBE
832-81-9528          992D6A5D91E7F59D03B940A4B1CBBCBE
```

Tip

If the target does not support binary data, use AES_ENCRYPT with the ENC_BASE64 function to store the data in a format compatible with the database.

ASCII

Availability:
Designer
Workflow Manager

When you configure the Integration Service to run in ASCII mode, the ASCII function returns the numeric ASCII value of the first character of the string passed to the function.

When you configure the Integration Service to run in Unicode mode, the ASCII function returns the numeric Unicode value of the first character of the string passed to the function. Unicode values fall in the range 0 to 65,535.

You can pass a string of any size to ASCII, but it evaluates only the first character in the string. Before you pass any string value to ASCII, you can parse out the specific character you want to convert to an ASCII or Unicode value. For example, you might use RTRIM or another string-manipulation function. If you pass a numeric value, ASCII converts it to a character string and returns the ASCII or Unicode value of the first character in the string.

This function is identical in behavior to the CHRCODE function. If you use ASCII in existing expressions, they will still work correctly. However, when you create new expressions, use the CHRCODE function instead of the ASCII function.

Syntax

```
ASCII ( string )
```

Argument	Required/ Optional	Description
string	Required	Character string. Passes the value you want to return as an ASCII value. You can enter any valid transformation expression.

Return Value

Integer. The ASCII or Unicode value of the first character in the string.

NULL if a value passed to the function is NULL.

Example

The following expression returns the ASCII or Unicode value for the first character of each value in the ITEMS port:

```
ASCII( ITEMS )
```

```
ITEMS                                                RETURN VALUE
Flashlight                                           70
```

ITEMS	RETURN VALUE
Compass	67
Safety Knife	83
Depth/Pressure Gauge	68
Regulator System	82

AVG

Availability:
Designer

Returns the average of all values in a group of rows. Optionally, you can apply a filter to limit the rows you read to calculate the average. You can nest only one other aggregate function within AVG, and the nested function must return a Numeric datatype.

Syntax

```
AVG( numeric_value [, filter_condition ] )
```

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. Passes the values for which you want to calculate an average. You can enter any valid transformation expression.
filter_condition	Optional	Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression.

Return Value

Numeric value.

NULL if all values passed to the function are NULL or if no rows are selected. For example, the filter condition evaluates to FALSE or NULL for all rows.

Note: If the return value is decimal with precision greater than 15, you can enable high precision in the session properties to ensure decimal precision up to 28 digits.

Nulls

If a value is NULL, AVG ignores the row. However, if all values passed from the port are NULL, AVG returns NULL.

Note: By default, the Integration Service treats null values as NULLs in aggregate functions. If you pass an entire port or group of null values, the function returns NULL. However, when you configure the Integration Service, you can choose how you want to handle null values in aggregate functions. You can treat null values as 0 in aggregate functions or as NULL.

Group By

AVG groups values based on group by ports you define in the transformation, returning one result for each group.

If there is not a group by port, AVG treats all rows as one group, returning one value.

Example

The following expression returns the average wholesale cost of flashlights:

```
AVG( WHOLESALE_COST, ITEM_NAME='Flashlight' )
```

ITEM_NAME	WHOLESALE_COST
Flashlight	35.00
Navigation Compass	8.05
Regulator System	150.00
Flashlight	29.00
Depth/Pressure Gauge	88.00
Flashlight	31.00

RETURN VALUE: 31.66

Tip

You can perform arithmetic on the values passed to AVG before the function calculates the average. For example:

```
AVG( QTY * PRICE - DISCOUNT )
```

CEIL

Availability:
Designer
Workflow Manager

Returns the smallest integer greater than or equal to the numeric value passed to this function. For example, if you pass 3.14 to CEIL, the function returns 4. If you pass 3.98 to CEIL, the function returns 4. Likewise, if you pass -3.17 to CEIL, the function returns -3.

Syntax

```
CEIL( numeric_value )
```

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. You can enter any valid transformation expression.

Return Value

Integer if you pass a numeric value with declared precision between 0 and 28.

Double value if you pass a numeric value with declared precision greater than 28.

NULL if a value passed to the function is NULL.

Example

The following expression returns the price rounded to the next integer:

```
CEIL( PRICE )
```

PRICE	RETURN VALUE
39.79	40
125.12	126

PRICE	RETURN VALUE
74.24	75
NULL	NULL
-100.99	-100

Tip: You can perform arithmetic on the values passed to CEIL before CEIL returns the next integer value. For example, if you wanted to multiply a numeric value by 10 before you calculated the smallest integer less than the modified value, you might write the function as follows:

```
CEIL( PRICE * 10 )
```

CHOOSE

Availability:
Designer
Workflow Manager

Chooses a string from a list of strings based on a given position. You specify the position and the value. If the value matches the position, the Integration Service returns the value.

Syntax

```
CHOOSE( index, string1 [, string2, ..., stringN] )
```

Argument	Required/ Optional	Description
index	Required	Numeric datatype. Enter a number based on the position of the value you want to match.
string	Required	Any character value.

Return Value

The string that matches the position of the index value.

NULL if no string matches the position of the index value.

Example

The following expression returns the string ‘flashlight’ based on an index value of 2:

```
CHOOSE( 2, 'knife', 'flashlight', 'diving hood' )
```

The following expression returns NULL based on an index value of 4:

```
CHOOSE( 4, 'knife', 'flashlight', 'diving hood' )
```

CHOOSE returns NULL because the expression does not contain a fourth argument.

CHR

Availability:
Designer
Workflow Manager

When you configure the Integration Service to move data in ASCII mode, CHR returns the ASCII character corresponding to the numeric value you pass to this function. ASCII values fall in the range 0 to 255. You can pass any integer to CHR, but only ASCII codes 32 to 126 are printable characters.

When you configure the Integration Service to move data in Unicode mode, CHR returns the Unicode character corresponding to the numeric value you pass to this function. Unicode values fall in the range 0 to 65,535.

Syntax

```
CHR( numeric_value )
```

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. The value you want to return as an ASCII or Unicode character. You can enter any valid transformation expression.

Return Value

ASCII or Unicode character. A string containing one character.

NULL if a value passed to the function is NULL.

Example

The following expression returns the ASCII or Unicode character for each numeric value in the ITEM_ID port:

```
CHR( ITEM_ID )
```

ITEM_ID	RETURN VALUE
65	A
122	z
NULL	NULL
88	X
100	d
71	G

Use the CHR function to concatenate a single quote onto a string. The single quote is the only character that you cannot use inside a string literal. Consider the following example:

```
'Joan' || CHR(39) || 's car'
```

The return value is:

```
Joan's car
```

CHRCODE

Availability:
Designer
Workflow Manager

When you configure the Integration Service to run in ASCII mode, CHRCODE returns the numeric ASCII value of the first character of the string passed to the function. ASCII values fall in the range 0 to 255.

When you configure the Integration Service to run in Unicode mode, CHRCODE returns the numeric Unicode value of the first character of the string passed to the function. Unicode values fall in the range 0 to 65,535.

Normally, before you pass any string value to CHRCODE, you parse out the specific character you want to convert to an ASCII or Unicode value. For example, you might use RTRIM or another string-manipulation function. If you pass a numeric value, CHRCODE converts it to a character string and returns the ASCII or Unicode value of the first character in the string.

This function is identical in behavior to the ASCII function. If you currently use ASCII in expressions, it will still work correctly. However, when you create new expressions, use the CHRCODE function instead of the ASCII function.

Syntax

```
CHRCODE ( string )
```

Argument	Required/ Optional	Description
string	Required	Character string. Passes the values you want to return as ASCII or Unicode values. You can enter any valid transformation expression.

Return Value

ASCII or Unicode character. A string containing one character.

NULL if a value passed to the function is NULL.

Example

The following expression returns the ASCII or Unicode value for the first character of each value in the ITEMS port:

```
CHRCODE( ITEMS )
```

ITEMS	RETURN VALUE
Flashlight	70
Compass	67
Safety Knife	83
Depth/Pressure Gauge	68
Regulator System	82

COMPRESS

Availability:
Designer
Workflow Manager

Compresses data using the zlib 1.2.1 compression algorithm. Use the COMPRESS function before you send large amounts of data over a wide area network.

Syntax

```
COMPRESS( value )
```

Argument	Required/ Optional	Description
value	Required	String datatype. Data that you want to compress.

Return Value

Compressed binary value of the input value.

NULL if the input is a null value.

Example

Your organization has an online order service. You want to send customer order data over a wide area network. The source contains a row that is 10 MB. You can compress the data in this row using COMPRESS. When you compress the data, you decrease the amount of data the Integration Service writes over the network. As a result, you may increase session performance.

CONCAT

Availability:
Designer
Workflow Manager

Concatenates two strings. CONCAT converts all data to text before concatenating the strings. Alternatively, use the || string operator to concatenate strings. Using the || string operator instead of CONCAT improves Integration Service performance when you run sessions.

Syntax

```
CONCAT( first_string, second_string )
```

Argument	Required/ Optional	Description
first_string	Required	Any datatype except Binary. The first part of the string you want to concatenate. You can enter any valid transformation expression.
second_string	Required	Any datatype except Binary. The second part of the string you want to concatenate. You can enter any valid transformation expression.

Return Value

String.

NULL if both string values are NULL.

Nulls

If one of the strings is NULL, CONCAT ignores it and returns the other string.

If both strings are NULL, CONCAT returns NULL.

Example

The following expression concatenates the names in the FIRST_NAME and LAST_NAME ports:

```
CONCAT( FIRST_NAME, LAST_NAME )
```

FIRST_NAME	LAST_NAME	RETURN VALUE
John	Baer	JohnBaer
NULL	Campbell	Campbell
Bobbi	Apperley	BobbiApperley
Jason	Wood	JasonWood
Dan	Covington	DanCovington
Greg	NULL	Greg
NULL	NULL	NULL
100	200	100200

CONCAT does not add spaces to separate strings. If you want to add a space between two strings, you can write an expression with two nested CONCAT functions. For example, the following expression first concatenates a space on the end of the first name and then concatenates the last name:

```
CONCAT( CONCAT( FIRST_NAME, ' ' ), LAST_NAME )
```

FIRST_NAME	LAST_NAME	RETURN VALUE
John	Baer	John Baer
NULL	Campbell	Campbell *(includes leading blank)*
Bobbi	Apperley	Bobbi Apperley
Jason	Wood	Jason Wood
Dan	Covington	Dan Covington
Greg	NULL	Greg
NULL	NULL	NULL

Use the CHR and CONCAT functions to concatenate a single quote onto a string. The single quote is the only character you cannot use inside a string literal. Consider the following example:

```
CONCAT( 'Joan', CONCAT( CHR(39), 's car' ))
```

The return value is:

```
Joan's car
```

CONVERT_BASE

Availability:
Designer
Workflow Manager

Converts a number from one base value to another base value.

Syntax

```
CONVERT_BASE( value, source_base, dest_base )
```

Argument	Required/ Optional	Description
value	Required	String datatype. Value you want to convert from one base to another base. Maximum is 9,233,372,036,854,775,806.
source_base	Required	Numeric datatype. Current base value of the data you want to convert. Minimum base is 2. Maximum base is 36.
dest_base	Required	Numeric datatype. Base value you want to convert the data to. Minimum base is 2. Maximum base is 36.

Return Value

Numeric value.

Example

The following example converts 2222 from the decimal base value 10 to the binary base value 2:

```
CONVERT_BASE( "2222", 10, 2 )
```

The Integration Service returns 100010101110.

COS

Availability:
Designer
Workflow Manager

Returns the cosine of a numeric value (expressed in radians).

Syntax

```
COS( numeric_value )
```

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. Numeric data expressed in radians (degrees multiplied by pi divided by 180). Passes the values for which you want to calculate a cosine. You can enter any valid transformation expression.

Return Value

Double value.

NULL if a value passed to the function is NULL.

Example

The following expression returns the cosine for all values in the Degrees port:

```
COS( DEGREES * 3.14159265359 / 180 )
```

DEGREES	RETURN VALUE
0	1.0
90	0.0
70	0.342020143325593
30	0.866025403784421
5	0.996194698091745
18	0.951056516295147
89	0.0174524064371813
NULL	NULL

Tip: You can perform arithmetic on the values passed to COS before the function calculates the cosine. For example, you can convert the values in the port to radians before calculating the cosine, as follows:

```
COS( ARCS * 3.14159265359 / 180 )
```

COSH

Availability:
Designer
Workflow Manager

Returns the hyperbolic cosine of a numeric value (expressed in radians).

Syntax

```
COSH( numeric_value )
```

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. Numeric data expressed in radians (degrees multiplied by pi divided by 180). Passes the values for which you want to calculate the hyperbolic cosine. You can enter any valid transformation expression.

Return Value

Double value.

NULL if a value passed to the function is NULL.

Example

The following expression returns the hyperbolic cosine for the values in the Angles port:

```
COSH( ANGLES )
```

ANGLES	RETURN VALUE
1.0	1.54308063481524
2.897	9.0874465864177
3.66	19.4435376920294
5.45	116.381231106176
0	1.0
0.345	1.06010513656773
NULL	NULL

Tip: You can perform arithmetic on the values passed to COSH before the function calculates the hyperbolic cosine. For example:

```
COSH( MEASURES.ARCS / 360 )
```

COUNT

Availability:
Designer

Returns the number of rows that have non-null values in a group. Optionally, you can include the asterisk (*) argument to count all input values in a transformation. You can nest only one other aggregate function within COUNT. You can apply a condition to filter rows before counting them.

Syntax

```
COUNT( value [, filter_condition] )
```

or

```
COUNT( * [, filter_condition] )
```

Argument	Required/ Optional	Description
value	Required	Any datatype except Binary. Passes the values you want to count. You can enter any valid transformation expression.
*	Optional	Use to count *all rows* in a transformation.
filter_condition	Optional	Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression.

Return Value

Integer.

0 if all values passed to this function are NULL (unless you include the asterisk argument).

Nulls

If all values are NULL, the function returns 0.

If you apply the asterisk argument, this function counts all rows, regardless if a column in a row contains a null value.

If you apply the *value* argument, this function ignores columns with null values.

Note: By default, the Integration Service treats null values as NULLs in aggregate functions. If you pass an entire port or group of null values, the function returns NULL. However, when you configure the Integration Service, you can choose how you want to handle null values in aggregate functions. You can treat null values as 0 in aggregate functions or as NULL.

Group By

COUNT groups values based on group by ports you define in the transformation, returning one result for each group. If there is no group by port COUNT treats all rows as one group, returning one value.

Examples

The following expression counts the items with less than 5 quantity in stock, excluding null values:

```
COUNT( ITEM_NAME, IN_STOCK < 5 )
```

ITEM_NAME	IN_STOCK
Flashlight	10
NULL	2
Compass	NULL
Regulator System	5
Safety Knife	8
Halogen Flashlight	1

RETURN VALUE: 1

In this example, the function counted the Halogen flashlight but not the NULL item. The function counts all rows in a transformation, including null values, as illustrated in the following example:

```
COUNT( *, QTY < 5 )
```

ITEM_NAME	QTY
Flashlight	10
NULL	2
Compass	NULL

```
ITEM_NAME                                                        QTY
Regulator System                                                 5
Safety Knife                                                     8
Halogen Flashlight                                               1
RETURN VALUE: 2
```

In this example, the function counts the NULL item and the Halogen Flashlight. If you include the asterisk argument, but do not use a filter, the function counts all rows that pass into the transformation. For example:

```
COUNT( * )
```

```
ITEM_NAME                                                        QTY
Flashlight                                                       10
NULL                                                             2
Compass                                                          NULL
Regulator System                                                 5
Safety Knife                                                     8
Halogen Flashlight                                               1
RETURN VALUE: 6
```

CRC32

Availability:
Designer
Workflow Manager

Returns a 32-bit Cyclic Redundancy Check (CRC32) value. Use CRC32 to find data transmission errors. You can also use CRC32 if you want to verify that data stored in a file has not been modified. If you use CRC32 to perform a redundancy check on data in ASCII mode and Unicode mode, the Integration Service may generate different results on the same input value.

Note: CRC32 can return the same output for different input strings. If you want to generate keys in a mapping, use a Sequence Generator transformation. If you use CRC32 to generate keys in a mapping, you may receive unexpected results.

Syntax

```
CRC32( value )
```

Argument	Required/ Optional	Description
value	Required	String or Binary datatype. Passes the values you want to perform a redundancy check on. Input value is case sensitive. The case of the input value affects the return value. For example, CRC32(informatica) and CRC32 (Informatica) return different values.

Return Value

32-bit integer value.

Example

You want to read data from a source across a wide area network. You want to make sure the data has been modified during transmission. You can compute the checksum for the data in the file and store it along with the

file. When you read the source data in PowerCenter, the Integration Service can use CRC32 to compute the checksum and compare it to the stored value. If the two values are the same, the data has not been modified.

CUME

Availability:
Designer

Returns a running total. A running total means CUME returns a total each time it adds a value. You can add a condition to filter rows out of the row set before calculating the running total.

Use CUME and similar functions (such as MOVINGAVG and MOVINGSUM) to simplify reporting by calculating running values.

Syntax

```
CUME( numeric_value [, filter_condition] )
```

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. Passes the values for which you want to calculate a running total. You can enter any valid transformation expression. You can create a nested expression to calculate a running total based on the results of the function as long as the result is a numeric value.
filter_condition	Optional	Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression.

Return Value

Numeric value.

NULL if all values passed to the function are NULL, or if no rows are selected (for example, the filter condition evaluates to FALSE or NULL for all rows).

Note: If the return value is decimal with precision greater than 15, you can enable high precision in the session properties to ensure decimal precision up to 28 digits.

Nulls

If a value is NULL, CUME returns the running total for the previous row. However, if all values in the selected port are NULL, CUME returns NULL.

Examples

The following sample rowset might result from using the CUME function:

```
CUME( PERSONAL_SALES )
```

PERSONAL_SALES	RETURN VALUE
40000	40000
80000	120000
40000	160000
60000	220000

PERSONAL_SALES	RETURN VALUE
NULL	220000
50000	270000

Likewise, you can add values before calculating a running total:

```
CUME( CA_SALES + OR_SALES )
```

CA_SALES	OR_SALES	RETURN VALUE
40000	10000	50000
80000	50000	180000
40000	2000	222000
60000	NULL	222000
NULL	NULL	222000
50000	3000	275000

DATE_COMPARE

Availability:
Designer
Workflow Manager

Returns an integer indicating which of two dates is earlier. DATE_COMPARE returns an integer value rather than a date value.

Syntax

```
DATE_COMPARE( date1, date2 )
```

Argument	Required/ Optional	Description
date1	Required	Date/Time datatype. The first date you want to compare. You can enter any valid transformation expression as long as it evaluates to a date.
date2	Required	Date/Time datatype. The second date you want to compare. You can enter any valid transformation expression as long as it evaluates to a date.

Return Value

-1 if the first date is earlier.

0 if the two dates are equal.

1 if the second date is earlier.

NULL if one of the date values is NULL.

Example

The following expression compares each date in the DATE_PROMISED and DATE_SHIPPED ports, and returns an integer indicating which date is earlier:

```
DATE_COMPARE( DATE_PROMISED, DATE_SHIPPED )
```

DATE_PROMISED	DATE_SHIPPED	RETURN VALUE
Jan 1 1997	Jan 13 1997	-1
Feb 1 1997	Feb 1 1997	0
Dec 22 1997	Dec 15 1997	1
Feb 29 1996	Apr 12 1996	-1 *(Leap year)*
NULL	Jan 6 1997	NULL
Jan 13 1997	NULL	NULL

DATE_DIFF

Availability:
Designer
Workflow Manager

Returns the length of time between two dates. You can request the format to be years, months, days, hours, minutes, seconds, milliseconds, microseconds, or nanoseconds. The Integration Service subtracts the second date from the first date and returns the difference.

Syntax

```
DATE_DIFF( date1, date2, format )
```

Argument	Required/ Optional	Description
date1	Required	Date/Time datatype. Passes the values for the first date you want to compare. You can enter any valid transformation expression.
date2	Required	Date/Time datatype. Passes the values for the second date you want to compare. You can enter any valid transformation expression.
format	Required	Format string specifying the date or time measurement. You can specify years, months, days, hours, minutes, seconds, milliseconds, microseconds, or nanoseconds. You can specify only one part of the date, such as 'mm'. Enclose the format strings within single quotation marks. The format string is not case sensitive. For example, the format string 'mm' is the same as 'MM', 'Mm' or 'mM'.

Return Value

Double value. If *date1* is later than *date2*, the return value is a positive number. If *date1* is earlier than *date2*, the return value is a negative number.

0 if the dates are the same.

NULL if one (or both) of the date values is NULL.

Examples

The following expressions return the number of hours between the DATE_PROMISED and DATE_SHIPPED ports:

```
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'HH' )
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'HH12' )
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'HH24' )
```

DATE_PROMISED	DATE_SHIPPED	RETURN VALUE
Jan 1 1997 12:00:00AM	Mar 29 1997 12:00:00PM	-2100
Mar 29 1997 12:00:00PM	Jan 1 1997 12:00:00AM	2100
NULL	Dec 10 1997 5:55:10PM	NULL
Dec 10 1997 5:55:10PM	NULL	NULL
Jun 3 1997 1:13:46PM	Aug 23 1996 4:20:16PM	6812.89166666667
Feb 19 2004 12:00:00PM	Feb 19 2005 12:00:00PM	-8784

The following expressions return the number of days between the DATE_PROMISED and the DATE_SHIPPED ports:

```
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'D' )
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'DD' )
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'DDD' )
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'DY' )
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'DAY' )
```

DATE_PROMISED	DATE_SHIPPED	RETURN VALUE
Jan 1 1997 12:00:00AM	Mar 29 1997 12:00:00PM	-87.5
Mar 29 1997 12:00:00PM	Jan 1 1997 12:00:00AM	87.5
NULL	Dec 10 1997 5:55:10PM	NULL
Dec 10 1997 5:55:10PM	NULL	NULL
Jun 3 1997 1:13:46PM	Aug 23 1996 4:20:16PM	283.870486111111
Feb 19 2004 12:00:00PM	Feb 19 2005 12:00:00PM	-366

The following expressions return the number of months between the DATE_PROMISED and DATE_SHIPPED ports:

```
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'MM' )
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'MON' )
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'MONTH' )
```

DATE_PROMISED	DATE_SHIPPED	RETURN VALUE
Jan 1 1997 12:00:00AM	Mar 29 1997 12:00:00PM	-2.91935483870968
Mar 29 1997 12:00:00PM	Jan 1 1997 12:00:00AM	2.91935483870968
NULL	Dec 10 1997 5:55:10PM	NULL
Dec 10 1997 5:55:10PM	NULL	NULL
Jun 3 1997 1:13:46PM	Aug 23 1996 4:20:16PM	9.3290162037037
Feb 19 2004 12:00:00PM	Feb 19 2005 12:00:00PM	-12

The following expressions return the number of years between the DATE_PROMISED and DATE_SHIPPED ports:

```
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'Y' )
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'YY' )
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'YYY' )
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'YYYY' )
```

DATE_PROMISED	DATE_SHIPPED	RETURN VALUE
Jan 1 1997 12:00:00AM	Mar 29 1997 12:00:00PM	-0.24327956989247
Mar 29 1997 12:00:00PM	Jan 1 1997 12:00:00AM	0.24327956989247
NULL	Dec 10 1997 5:55:10PM	NULL
Dec 10 1997 5:55:10PM	NULL	NULL
Jun 3 1997 1:13:46PM	Aug 23 1996 4:20:16PM	0.77741801697531
Feb 19 2004 12:00:00PM	Feb 19 2005 12:00:00PM	-1

DEC_BASE64

Availability:
Designer
Workflow Manager

Decodes a base 64 encoded value and returns a string with the binary data representation of the data. If you encode data using ENC_BASE64, and you want to decode data using DEC_BASE64, you must run the decoding session using the same data movement mode. Otherwise, the output of the decoded data may differ from the original data.

Syntax

```
DEC_BASE64( value )
```

Argument	Required/ Optional	Description
value	Required	String datatype. Data that you want to decode.

Return Value

Binary decoded value.

NULL if the input is a null value.

Return values differ if you run the session in Unicode mode versus ASCII mode.

Example

You encoded WebSphere MQ message IDs and wrote them to a flat file during a workflow. You want to read data from the flat file source, including the WebSphere MQ message IDs. You can use DEC_BASE64 to decode the IDs and convert them to their original binary value.

DECODE

Availability:
Designer
Workflow Manager

Searches a port for a value you specify. If the function finds the value, it returns a result value, which you define. You can build an unlimited number of searches within a DECODE function.

If you use DECODE to search for a value in a string port, you can either trim trailing blanks with the RTRIM function or include the blanks in the search string.

Syntax

```
DECODE( value, first_search, first_result [, second_search, second_result]...[,default] )
```

Argument	Required/ Optional	Description
value	Required	Any datatype except Binary. Passes the values you want to search. You can enter any valid transformation expression.
search	Required	Any value with the same datatype as the value argument. Passes the values for which you want to search. The search value must match the value argument. You cannot search for a portion of a value. Also, the search value is case sensitive. For example, if you want to search for the string 'Halogen Flashlight' in a particular port, you must enter 'Halogen Flashlight, not just 'Halogen'. If you enter 'Halogen', the search does not find a matching value. You can enter any valid transformation expression.
result	Required	Any datatype except Binary. The value you want to return if the search finds a matching value. You can enter any valid transformation expression.
default	Optional	Any datatype except Binary. The value you want to return if the search does not find a matching value. You can enter any valid transformation expression.

Return Value

First_result if the search finds a matching value.

Default value if the search does not find a matching value.

NULL if you omit the default argument and the search does not find a matching value.

Even if multiple conditions are met, the Integration Service returns the first matching result.

If the data contains multibyte characters and the DECODE expression compares string data, the return value depends on the code page and data movement mode of the Integration Service.

DECODE and Datatypes

When you use DECODE, the datatype of the return value is always the same as the datatype of the result with the greatest precision.

For example, you have the following expression:

```
DECODE ( CONST_NAME
         'Five', 5,
         'Pythagoras', 1.414213562,
         'Archimedes', 3.141592654,
         'Pi', 3.141592654 )
```

The return values in this expression are 5, 1.414213562, and 3.141592654. The first result is an Integer, and the other results are Decimal. The Decimal datatype has greater precision than Integer. This expression always writes the result as a Decimal.

When you run a session in high precision mode, if at least one result is Double, the datatype of the return value is Double.

You cannot create a DECODE function with both string and numeric return values.

For example, the following expression is invalid:

```
DECODE ( CONST_NAME
         'Five', 5,
         'Pythagoras', '1.414213562',
         'Archimedes', '3.141592654',
         'Pi', 3.141592654 )
```

When you validate the expression above, the Designer writes the following error message:

```
Function cannot resolve operands of ambiguously mismatching datatypes.
```

Examples

You might use DECODE in an expression that searches for a particular ITEM_ID and returns the ITEM_NAME:

```
DECODE( ITEM_ID, 10, 'Flashlight',
                 14, 'Regulator',
                 20, 'Knife',
                 40, 'Tank',
                 'NONE' )
```

ITEM_ID	RETURN VALUE
10	Flashlight
14	Regulator
17	NONE
20	Knife
25	NONE
NULL	NONE
40	Tank

DECODE returns the default value of NONE for items 17 and 25 because the search values did not match the ITEM_ID. Also, DECODE returns NONE for the NULL ITEM_ID.

The following expression tests multiple columns and conditions, evaluated in a top to bottom order for TRUE or FALSE:

```
DECODE( TRUE,
        Var1 = 22, 'Variable 1 was 22!',
        Var2 = 49, 'Variable 2 was 49!',
        Var1 < 23, 'Variable 1 was less than 23.',
        Var2 > 30, 'Variable 2 was more than 30.',
        'Variables were out of desired ranges.')
```

Var1	Var2	RETURN VALUE
21	47	Variable 1 was less than 23.
22	49	Variable 1 was 22!
23	49	Variable 2 was 49!
24	27	Variables were out of desired ranges.
25	50	Variable 2 was more than 30.

DECOMPRESS

Availability:
Designer
Workflow Manager

Decompresses data using the zlib 1.2.1 compression algorithm. Use the DECOMPRESS function on data that has been compressed with the COMPRESS function or a compression tool that uses the zlib 1.2.1 algorithm. If the session that decompresses the data uses a different data movement mode than the session that compressed the data, the output of the decompressed data may differ from the original data.

Syntax

```
DECOMPRESS( value, precision )
```

Argument	Required/ Optional	Description
value	Required	Binary datatype. Data that you want to decompress.
precision	Optional	Integer datatype.

Return Value

Decompressed binary value of the input value.

NULL if the input is a null value.

Example

Your organization has an online order service. You received compressed customer order data over a wide area network. You want to read the data using PowerCenter and load the data to a data warehouse. You can decompress each row of data using DECOMPRESS for the row. The Integration Service can then load the decompressed data to the target.

ENC_BASE64

Availability:
Designer
Workflow Manager

Encodes data by converting binary data to string data using Multipurpose Internet Mail Extensions (MIME) encoding. Encode data when you want to store data in a database or file that does not allow binary data. You can also encode data to pass binary data through PowerCenter transformations in string format. The encoded data is approximately 33% longer than the original data. It displays as a set of random characters.

Syntax

```
ENC_BASE64( value )
```

Argument	Required/ Optional	Description
value	Required	Binary or String datatype. Data that you want to encode.

Return Value

Encoded value.

NULL if the input is a null value.

Example

You want to read messages from WebSphere MQ and write the data to a flat file target. You want to include the WebSphere MQ message ID as part of the target data. However, the MsgID field is Binary, and the flat file target

does not support binary data. Use ENC_BASE64 to encode the MsgID before the Integration Service writes the data to the target.

ERROR

Availability:
Designer

Causes the Integration Service to skip a row and issue an error message, which you define. The error message displays in the session log. The Integration Service does not write these skipped rows to the session reject file.

Use ERROR in Expression transformations to validate data. Generally, you use ERROR within an IIF or DECODE function to set rules for skipping rows.

Use the ERROR function for both input and output port default values. You might use ERROR for input ports to keep null values from passing into a transformation.

Use ERROR for output ports to handle any kind of transformation error, including ERROR function calls within an expression. When you use the ERROR function in an expression and in the output port default value, the Integration Service skips the row and logs both the error message from the expression and the error message from the default value. If you want to ensure the Integration Service skips rows that produce an error, assign ERROR as the default value.

If you use an output default value other than ERROR, the default value overrides the ERROR function in an expression. For example, you use the ERROR function in an expression, and you assign the default value, ‘1234’, to the output port. Each time the Integration Service encounters the ERROR function in the expression, it overrides the error with the value ‘1234’ and passes ‘1234’ to the next transformation. It does not skip the row, and it does not log an error in the session log.

Syntax

```
ERROR( string )
```

Argument	Required/ Optional	Description
string	Required	String value. The message you want to display when the Integration Service skips a row based on the expression containing the ERROR function. The string can be any length.

Return Value

String.

Example

The following example shows how to reference a mapping that calculates the average salary for employees in all departments of the organization, but skip negative values. The following expression nests the ERROR function in an IIF expression so that if the Integration Service finds a negative salary in the Salary port, it skips the row and displays an error:

```
IIF( SALARY < 0, ERROR ('Error. Negative salary found. Row skipped.', EMP_SALARY )
```

```
SALARY      RETURN VALUE
10000       10000
-15000      'Error. Negative salary found. Row skipped.'
```

```
SALARY      RETURN VALUE
NULL        NULL
150000      150000
1005        1005
```

EXP

Availability:
Designer
Workflow Manager

Returns e raised to the specified power (exponent), where e=2.71828183. For example, EXP(2) returns 7.38905609893065. You might use this function to analyze scientific and technical data rather than business data. EXP is the reciprocal of the LN function, which returns the natural logarithm of a numeric value.

Syntax

```
EXP( exponent )
```

Argument	Required/ Optional	Description
exponent	Required	Numeric datatype. The value to which you want to raise e. The exponent in the equation e^*value*. You can enter any valid transformation expression.

Return Value

Double value.

NULL if a value passed as an argument to the function is NULL.

Example

The following expression uses the values stored in the Numbers port as the exponent value:

```
EXP( NUMBERS )
```

```
NUMBERS                 RETURN VALUE
10                      22026.4657948067
-2                      0.135335283236613
8.55                    5166.754427176
NULL                    NULL
```

FIRST

Availability:
Designer

Returns the first value found within a port or group. Optionally, you can apply a filter to limit the rows the Integration Service reads. You can nest only one other aggregate function within FIRST.

Syntax

```
FIRST( value [, filter_condition ] )
```

Argument	Required/ Optional	Description
value	Required	Any datatype except Binary. Passes the values for which you want to return the first value. You can enter any valid transformation expression.
filter_condition	Optional	Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression.

Return Value

First value in a group.

NULL if all values passed to the function are NULL or if no rows are selected (for example, the filter condition evaluates to FALSE or NULL for all rows).

Nulls

If a value is NULL, FIRST ignores the row. However, if all values passed from the port are NULL, FIRST returns NULL.

Note: By default, the Integration Service treats null values as NULLs in aggregate functions. If you pass an entire port or group of null values, the function returns NULL. However, when you configure the Integration Service, you can choose how you want to handle null values in aggregate functions. You can treat null values as 0 in aggregate functions or as NULL.

Group By

FIRST groups values based on group by ports you define in the transformation, returning one result for each group.

If there is no group by port, FIRST treats all rows as one group, returning one value.

Examples

The following expression returns the first value in the ITEM_NAME port with a price greater than $10.00:

```
FIRST( ITEM_NAME, ITEM_PRICE > 10 )
```

```
ITEM_NAME                                                ITEM_PRICE
Flashlight                                               35.00
Navigation Compass                                       8.05
Regulator System                                         150.00
Flashlight                                               29.00
Depth/Pressure Gauge                                     88.00
Flashlight                                               31.00
RETURN VALUE:  Flashlight
```

The following expression returns the first value in the ITEM_NAME port with a price greater than $40.00:

```
FIRST( ITEM_NAME, ITEM_PRICE > 40 )
```

```
ITEM_NAME                                                ITEM_PRICE
Flashlight                                               35.00
Navigation Compass                                       8.05
Regulator System                                         150.00
Flashlight                                               29.00
Depth/Pressure Gauge                                     88.00
Flashlight                                               31.00
RETURN VALUE:  Regulator System
```

FLOOR

Availability:
Designer
Workflow Manager

Returns the largest integer less than or equal to the numeric value you pass to this function. For example, if you pass 3.14 to FLOOR, the function returns 3. If you pass 3.98 to FLOOR, the function returns 3. Likewise, if you pass -3.17 to FLOOR, the function returns -4.

Syntax

```
FLOOR( numeric_value )
```

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. You can enter any valid transformation expression as long as it evaluates to numeric data.

Return Value

Integer if you pass a numeric value with declared precision between 0 and 28.

Double if you pass a numeric value with declared precision greater than 28.

NULL if a value passed to the function is NULL.

Example

The following expression returns the largest integer less than or equal to the values in the PRICE port:

```
FLOOR( PRICE )
```

PRICE	RETURN VALUE
39.79	39
125.12	125
74.24	74
NULL	NULL
-100.99	-101

Tip: You can perform arithmetic on the values you pass to FLOOR. For example, to multiply a numeric value by 10 and then calculate the largest integer that is less than the product, you might write the function as follows:

```
FLOOR( UNIT_PRICE * 10 )
```

FV

Availability:
Designer
Workflow Manager

Returns the future value of an investment, where you make periodic, constant payments and the investment earns a constant interest rate.

Syntax

```
FV( rate, terms, payment [, present value, type] )
```

Argument	Required/ Optional	Description
rate	Required	Numeric. Interest rate earned in each period. Expressed as a decimal number. Divide the percent rate by 100 to express it as a decimal number. Must be greater than or equal to 0.
terms	Required	Numeric. Number of periods or payments. Must be greater than 0.
payment	Required	Numeric. Payment amount due per period. Must be a negative number
present value	Optional	Numeric. Current value of the investment. If you omit this argument, FV uses 0.
type	Optional	Integer. Timing of the payment. Enter 1 if payment is at the beginning of period. Enter 0 if payment is at the end of period. Default is 0. If you enter a value other than 0 or 1, the Integration Service treats the value as 1.

Return Value

Numeric.

Example

You deposit $2,000 into an account that earns 9% annual interest compounded monthly (monthly interest of 9%/12, or 0.75%). You plan to deposit $250 at the beginning of every month for the next 12 months. The following expression returns $5,337.96 as the account balance at the end of 12 months:

```
FV(0.0075, 12, -250, -2000, TRUE)
```

Notes

To calculate interest rate earned in each period, divide the annual rate by the number of payments made in a year. The payment value and present value are negative because these are amounts that you pay.

GET_DATE_PART

Availability:
Designer
Workflow Manager

Returns the specified part of a date as an integer value. Therefore, if you create an expression that returns the month portion of the date, and pass a date such as Apr 1 1997 00:00:00, GET_DATE_PART returns 4.

Syntax

```
GET_DATE_PART( date, format )
```

Argument	Required/ Optional	Description
date	Required	Date/Time datatype. You can enter any valid transformation expression.
format	Required	A format string specifying the portion of the date value you want to return. Enclose format strings within single quotation marks, for example, 'mm'. The format string is not case sensitive. Each format string returns the entire part of the date based on the date format specified in the session. For example, if you pass the date Apr 1 1997 to GET_DATE_PART, the format strings 'Y', 'YY', 'YYY', or 'YYYY' all return 1997.

Return Value

Integer representing the specified part of the date.

NULL if a value passed to the function is NULL.

Examples

The following expressions return the hour for each date in the DATE_SHIPPED port. 12:00:00AM returns 0 because the default date format is based on the 24 hour interval:

```
GET_DATE_PART( DATE_SHIPPED, 'HH' )
GET_DATE_PART( DATE_SHIPPED, 'HH12' )
GET_DATE_PART( DATE_SHIPPED, 'HH24' )
```

DATE_SHIPPED	RETURN VALUE
Mar 13 1997 12:00:00AM	0
Sep 2 1997 2:00:01AM	2
Aug 22 1997 12:00:00PM	12
June 3 1997 11:30:44PM	23
NULL	NULL

The following expressions return the day for each date in the DATE_SHIPPED port:

```
GET_DATE_PART( DATE_SHIPPED, 'D' )
GET_DATE_PART( DATE_SHIPPED, 'DD' )
GET_DATE_PART( DATE_SHIPPED, 'DDD' )
GET_DATE_PART( DATE_SHIPPED, 'DY' )
GET_DATE_PART( DATE_SHIPPED, 'DAY' )
```

DATE_SHIPPED	RETURN VALUE
Mar 13 1997 12:00:00AM	13
June 3 1997 11:30:44PM	3
Aug 22 1997 12:00:00PM	22
NULL	NULL

The following expressions return the month for each date in the DATE_SHIPPED port:

```
GET_DATE_PART( DATE_SHIPPED, 'MM' )
GET_DATE_PART( DATE_SHIPPED, 'MON' )
GET_DATE_PART( DATE_SHIPPED, 'MONTH' )
```

DATE_SHIPPED	RETURN VALUE
Mar 13 1997 12:00:00AM	3
June 3 1997 11:30:44PM	6
NULL	NULL

The following expression return the year for each date in the DATE_SHIPPED port:

```
GET_DATE_PART( DATE_SHIPPED, 'Y' )
GET_DATE_PART( DATE_SHIPPED, 'YY' )
GET_DATE_PART( DATE_SHIPPED, 'YYY' )
GET_DATE_PART( DATE_SHIPPED, 'YYYY' )
```

DATE_SHIPPED	RETURN VALUE
Mar 13 1997 12:00:00AM	1997
June 3 1997 11:30:44PM	1997
NULL	NULL

GREATEST

Availability:
Designer
Workflow Manager

Returns the greatest value from a list of input values. Use this function to return the greatest string, date, or number. By default, the match is case sensitive.

Syntax

```
GREATEST( value1, [value2, ..., valueN,] CaseFlag )
```

Argument	Required/ Optional	Description
value	Required	Any datatype except Binary. Datatype must be compatible with other values. Value you want to compare against other values. You must enter at least one value argument. If the value is numeric, and other input values are numeric, all values use the highest precision possible. For example, if some values are Integer datatype and others are Double datatype, the Integration Service converts the values to Double.
CaseFlag	Optional	Must be an integer. Determines whether the arguments in this function are case sensitive. You can enter any valid transformation expression. When *CaseFlag* is a number other than 0, the function is case sensitive. When *CaseFlag* is a null value or 0, the function is not case sensitive.

Return Value

value1 if it is the greatest of the input values, *value2* if it is the greatest of the input values, and so on.

NULL if any of the arguments is null.

Example

The following expression returns the greatest quantity of items ordered:

```
GREATEST( QUANTITY1, QUANTITY2, QUANTITY3 )
```

QUANTITIY1	QUANTITY2	QUANTITY3	RETURN VALUE
150	756	27	756
			NULL
5000	97	17	5000
120	1724	965	1724

IIF

Availability:
Designer
Workflow Manager

Returns one of two values you specify, based on the results of a condition.

Syntax

```
IIF( condition, value1 [,value2] )
```

Argument	Required/ Optional	Description
condition	Required	The condition you want to evaluate. You can enter any valid transformation expression that evaluates to TRUE or FALSE.
value1	Required	Any datatype except Binary. The value you want to return if the condition is TRUE. The return value is always the datatype specified by this argument. You can enter any valid transformation expression, including another IIF expression.
value2	Optional	Any datatype except Binary. The value you want to return if the condition is FALSE. You can enter any valid transformation expression, including another IIF expression.

Unlike conditional functions in some systems, the FALSE (*value2*) condition in the IIF function is not required. If you omit *value2*, the function returns the following when the condition is FALSE:

- 0 if *value1* is a Numeric datatype.
- Empty string if *value1* is a String datatype.
- NULL if *value1* is a Date/Time datatype.

For example, the following expression does not include a FALSE condition and *value1* is a string datatype so the Integration Service returns an empty string for each row that evaluates to FALSE:

```
IIF( SALES > 100, EMP_NAME )
```

SALES	EMP_NAME	RETURN VALUE
150	John Smith	John Smith
50	Pierre Bleu	'' *(empty string)*
120	Sally Green	Sally Green
NULL	Greg Jones	'' *(empty string)*

Return Value

value1 if the condition is TRUE.

value2 if the condition is FALSE.

For example, the following expression includes the FALSE condition NULL so the Integration Service returns NULL for each row that evaluates to FALSE:

```
IIF( SALES > 100, EMP_NAME, NULL )
```

SALES	EMP_NAME	RETURN VALUE
150	John Smith	John Smith
50	Pierre Bleu	NULL

SALES	EMP_NAME	RETURN VALUE
120	Sally Green	Sally Green
NULL	Greg Jones	NULL

If the data contains multibyte characters and the condition argument compares string data, the return value depends on the code page and data movement mode of the Integration Service.

IIF and Datatypes

When you use IIF, the datatype of the return value is the same as the datatype of the result with the greatest precision.

For example, you have the following expression:

```
IIF( SALES < 100, 1, .3333 )
```

The TRUE result (1) is an integer and the FALSE result (.3333) is a decimal. The Decimal datatype has greater precision than Integer, so the datatype of the return value is always a Decimal.

When you run a session in high precision mode and at least one result is Double, the datatype of the return value is Double.

Special Uses of IIF

Use nested IIF statements to test multiple conditions. The following example tests for various conditions and returns 0 if sales is 0 or negative:

```
IIF( SALES > 0, IIF( SALES < 50, SALARY1, IIF( SALES < 100, SALARY2, IIF( SALES < 200, SALARY3,
BONUS))), 0 )
```

You can make this logic more readable by adding comments:

```
IIF( SALES > 0,
--then test to see if sales is between 1 and 49:
   IIF( SALES < 50,

   --then return SALARY1
      SALARY1,

      --else test to see if sales is between 50 and 99:
         IIF( SALES < 100,

          --then return
             SALARY2,

             --else test to see if sales is between 100 and 199:
                IIF( SALES < 200,

                --then return
                   SALARY3,

                  --else for sales over 199, return
                     BONUS)
            )
          ),

--else for sales less than or equal to zero, return
      0)
```

Use IIF in update strategies. For example:

```
IIF( ISNULL( ITEM_NAME ), DD_REJECT, DD_INSERT)
```

Alternative to IIF

Use “DECODE” on page 64 instead of IIF in many cases. DECODE may improve readability. The following shows how you use DECODE instead of IIF using the first example from the previous section:

```
DECODE( TRUE,
   SALES > 0 and SALES < 50, SALARY1,
   SALES > 49 AND SALES < 100, SALARY2,
```

```
SALES > 99 AND SALES < 200, SALARY3,
SALES > 199, BONUS)
```

You can often use a Filter transformation instead of IIF to maximize session performance.

IN

Availability:
Designer
Workflow Manager

Matches input data to a list of values. By default, the match is case sensitive.

Syntax

```
IN( valueToSearch, value1, [value2, ..., valueN,] CaseFlag )
```

Argument	Required/ Optional	Description
valueToSearch	Required	Can be a string, date, or numeric value. Input value you want to match against a comma-separated list of values.
value	Required	Can be a string, date, or numeric value. Comma-separated list of values you want to search for. Values can be ports in a transformation. There is no maximum number of values you can list.
CaseFlag	Optional	Must be an integer. Determines whether the arguments in this function are case sensitive. You can enter any valid transformation expression. When *CaseFlag* is a number other than 0, the function is case sensitive. When *CaseFlag* is a null value or 0, the function is not case sensitive.

Return Value

TRUE (1) if the input value matches the list of values.

FALSE (0) if the input value does not match the list of values.

NULL if the input is a null value.

Example

The following expression determines if the input value is a safety knife, chisel point knife, or medium titanium knife. The input values do not have to match the case of the values in the comma-separated list:

```
IN( ITEM_NAME, 'Chisel Point Knife', 'Medium Titanium Knife', 'Safety Knife', 0 )
```

ITEM_NAME	RETURN VALUE
Stabilizing Vest	0 (FALSE)
Safety knife	1 (TRUE)
Medium Titanium knife	1 (TRUE)
	NULL

INDEXOF

Availability:
Designer
Workflow Manager

Finds the index of a value among a list of values. By default, the match is case sensitive.

Syntax

```
INDEXOF( valueToSearch, string1, [string2, ..., stringN,] CaseFlag )
```

Argument	Required/ Optional	Description
valueToSearch	Required	String datatype. Value you want to search for in the list of strings.
string	Required	String datatype. Comma-separated list of values you want to search against. Values can be in string format. There is no maximum number of values you can list. The value is case sensitive, unless you set MatchCase to 0.
CaseFlag	Required	Must be an integer. Determines whether the arguments in this function are case sensitive. You can enter any valid transformation expression. When *CaseFlag* is a number other than 0, the function is case sensitive. When *CaseFlag* is a null value or 0, the function is not case sensitive.

Return Value

1 if the input value matches *string1*, 2 if the input value matches *string2*, and so on.

0 if the input value is not found.

NULL if the input is a null value.

Example

The following expression determines if values from the ITEM_NAME port match the first, second, or third string:

```
INDEXOF( ITEM_NAME, 'diving hood', 'flashlight', 'safety knife')
```

ITEM_NAME	RETURN VALUE
Safety Knife	0
diving hood	1
Compass	0
safety knife	3
flashlight	2

Safety Knife returns a value of 0 because it does not match the case of the input value.

INITCAP

Availability:
Designer
Workflow Manager

Capitalizes the first letter in each word of a string and converts all other letters to lowercase. Words are delimited by white space (a blank space, formfeed, newline, carriage return, tab, or vertical tab) and characters that are not alphanumeric. For example, if you pass the string '…THOMAS', the function returns Thomas.

Syntax

```
INITCAP( string )
```

Argument	Required/ Optional	Description
string	Required	Any datatype except Binary. You can enter any valid transformation expression.

Return Value

String. If the data contains multibyte characters, the return value depends on the code page and data movement mode of the Integration Service.

NULL if a value passed to the function is NULL.

Example

The following expression capitalizes all names in the FIRST_NAME port.

```
INITCAP( FIRST_NAME )
```

FIRST_NAME	RETURN VALUE
ramona	Ramona
18-albert	18-Albert
NULL	NULL
?!SAM	?!Sam
THOMAS	Thomas
PierRe	Pierre

INSTR

Availability:
Designer
Workflow Manager

Returns the position of a character set in a string, counting from left to right.

Syntax

```
INSTR( string, search_value [,start [,occurrence [,comparison_type ]]] )
```

Argument	Required/ Optional	Description
string	Required	The string must be a character string. Passes the value you want to evaluate. You can enter any valid transformation expression. The results of the expression must be a character string. If not, INSTR converts the value to a string before evaluating it.
search_value	Required	Any value. The search value is case sensitive. The set of characters you want to search for. The search_value must match a part of the string. For example, if you write `INSTR('Alfred Pope', 'Alfred Smith')` the function returns 0. You can enter any valid transformation expression. If you want to search for a character string, enclose the characters you want to search for in single quotation marks, for example 'abc'.
start	Optional	Must be an integer value. The position in the string where you want to start the search. You can enter any valid transformation expression. The default is 1, meaning that INSTR starts the search at the first character in the string. If the start position is 0, INSTR searches from the first character in the string. If the start position is a positive number, INSTR locates the start position by counting from the beginning of the string. If the start position is a negative number, INSTR locates the start position by counting from the end of the string. If you omit this argument, the function uses the default value of 1.
occurrence	Optional	A positive integer greater than 0. You can enter any valid transformation expression. If the search value appears more than once in the string, you can specify which occurrence you want to search for. For example, you would enter 2 to search for the second occurrence from the start position. If you omit this argument, the function uses the default value of 1, meaning that INSTR searches for the first occurrence of the search value. If you pass a decimal, the Integration Service rounds it to the nearest integer value. If you pass a negative integer or 0, the session fails.
comparison_type	Optional	The string comparison type, either linguistic or binary, when the Integration Service runs in Unicode mode. When the Integration Service runs in ASCII mode, the comparison type is always binary. Linguistic comparisons take language-specific collation rules into account, while binary comparisons perform bitwise matching. For example, the German sharp s character matches the string “ss” in a linguistic comparison, but not in a binary comparison. Binary comparisons run faster than linguistic comparisons. Must be an integer value, either 0 or 1: - 0: INSTR performs a linguistic string comparison. - 1: INSTR performs a binary string comparison. Default is 0.

Return Value

Integer if the search is successful. Integer represents the position of the first character in the *search_value*, counting from left to right.

0 if the search is unsuccessful.

NULL if a value passed to the function is NULL.

Examples

The following expression returns the position of the first occurrence of the letter ‘a’, starting at the beginning of each company name. Because the *search_value* argument is case sensitive, it skips the ‘A’ in ‘Blue Fin Aqua Center’, and returns the position for the ‘a’ in ‘Aqua’:

```
INSTR( COMPANY, 'a' )
```

COMPANY	RETURN VALUE
Blue Fin Aqua Center	13
Maco Shark Shop	2
Scuba Gear	5
Frank's Dive Shop	3
VIP Diving Club	0

The following expression returns the position of the second occurrence of the letter ‘a’, starting at the beginning of each company name. Because the *search_value* argument is case sensitive, it skips the ‘A’ in ‘Blue Fin Aqua Center’, and returns 0:

```
INSTR( COMPANY, 'a', 1, 2 )
```

COMPANY	RETURN VALUE
Blue Fin Aqua Center	0
Maco Shark Shop	8
Scuba Gear	9
Frank's Dive Shop	0
VIP Diving Club	0

The following expression returns the position of the second occurrence of the letter ‘a’ in each company name, starting from the last character in the company name. Because the *search_value* argument is case sensitive, it skips the ‘A’ in 'Blue Fin Aqua Center’, and returns 0:

```
INSTR( COMPANY, 'a', -1, 2 )
```

COMPANY	RETURN VALUE
Blue Fin Aqua Center	0
Maco Shark Shop	2
Scuba Gear	5
Frank's Dive Shop	0
VIP Diving Club	0

The following expression returns the position of the first character in the string ‘Blue Fin Aqua Center’ (starting from the last character in the company name):

```
INSTR( COMPANY, 'Blue Fin Aqua Center', -1, 1 )
```

COMPANY	RETURN VALUE
Blue Fin Aqua Center	1
Maco Shark Shop	0
Scuba Gear	0
Frank's Dive Shop	0
VIP Diving Club	0

Using Nested INSTR

You can nest the INSTR function within other functions to accomplish more complex tasks.

The following expression evaluates a string, starting from the end of the string. The expression finds the last (rightmost) space in the string and then returns all characters to the left of it:

```
SUBSTR( CUST_NAME,1,INSTR( CUST_NAME,' ' ,-1,1 ))
```

CUST_NAME	RETURN VALUE
PATRICIA JONES	PATRICIA
MARY ELLEN SHAH	MARY ELLEN

The following expression removes the character '#' from a string:

```
SUBSTR( CUST_ID, 1, INSTR(CUST_ID, '#')-1 ) || SUBSTR( CUST_ID, INSTR(CUST_ID, '#')+1 )
```

CUST_ID	RETURN VALUE
ID#33	ID33
#A3577	A3577
SS #712403399	SS 712403399

ISNULL

Availability:
Designer
Workflow Manager

Returns whether a value is NULL. ISNULL evaluates an empty string as FALSE.

Note: To test for empty strings, use LENGTH.

Syntax

```
ISNULL( value )
```

Argument	Required/ Optional	Description
value	Required	Any datatype except Binary. Passes the rows you want to evaluate. You can enter any valid transformation expression.

Return Value

TRUE (1) if the value is NULL.

FALSE (0) if the value is not NULL.

Example

The following example checks for null values in the items table:

```
ISNULL( ITEM_NAME )
```

ITEM_NAME	RETURN VALUE
Flashlight	0 (FALSE)
NULL	1 (TRUE)
Regulator system	0 (FALSE)
''	0 (FALSE) *Empty string is not NULL*

IS_DATE

Availability:
Designer
Workflow Manager

Returns whether a string value is a valid date. A valid date is any string in the date portion of the date time format specified in the session. If the string you want to test is not in this date format, use the TO_DATE format string to specify the date format. If the strings passed to IS_DATE do not match the format string specified, the function returns FALSE (0). If the strings match the format string, the function returns TRUE (1).

IS_DATE evaluates strings and returns an integer value.

The output port for an IS_DATE expression must be String or Numeric datatype.

You might use IS_DATE to test or filter data in a flat file before writing it to a target.

Use the RR format string with IS_DATE instead of the YY format string. In most cases, the two format strings return the same values, but there are some unique cases where YY returns incorrect results. For example, the expression IS_DATE('02/29/00', 'YY') is internally computed as IS_DATE(02/29/1900 00:00:00), which returns false. However, the Integration Service computes the expression IS_DATE('02/29/00', 'RR') as IS_DATE(02/29/2000 00:00:00), which returns TRUE. In the first case, year 1900 is not a leap year, so there is no February 29th.

Note: IS_DATE uses the same format strings as TO_DATE.

Syntax

```
IS_DATE( value [,format] )
```

Argument	Required/ Optional	Description
value	Required	Must be a string datatype. Passes the rows you want to evaluate. You can enter any valid transformation expression.
format	Optional	Enter a valid TO_DATE format string. The format string must match the parts of the *string* argument. For example, if you pass the string 'Mar 15 1997 12:43:10AM', you must use the format string 'MON DD YYYY HH12:MI:SSAM'. If you omit the format string, the string value must be in the date format specified in the session.

Return Value

TRUE (1) if the row is a valid date.

FALSE (0) if the row is not a valid date.

NULL if a value in the expression is NULL or if the format string is NULL.

Warning: The format of the IS_DATE string must match the format string, including any date separators. If it does not, the Integration Service might return inaccurate values or skip the record.

Examples

The following expression checks the INVOICE_DATE port for valid dates:

```
IS_DATE( INVOICE_DATE )
```

This expression returns data similar to the following:

```
INVOICE_DATE                    RETURN VALUE
NULL                            NULL
'180'                           0 (FALSE)
'04/01/98'                      0 (FALSE)
'04/01/1998 00:12:15.7008'      1 (TRUE)
'02/31/1998 12:13:55.9204'      0 (FALSE)   (February does not have 31 days)
'John Smith'                    0 (FALSE)
```

The following IS_DATE expression specifies a format string of 'YYYY/MM/DD':

```
IS_DATE( INVOICE_DATE, 'YYYY/MM/DD' )
```

If the string value does not match this format, IS_DATE returns FALSE:

```
INVOICE_DATE                    RETURN VALUE
NULL                            NULL
'180'                           0 (FALSE)
'04/01/98'                      0 (FALSE)
'1998/01/12'                    1 (TRUE)
'1998/11/21 00:00:13'           0 (FALSE)
'1998/02/31'                    0 (FALSE)   (February does not have 31 days)
'John Smith'                    0 (FALSE)
```

The following example shows how you use IS_DATE to test data before using TO_DATE to convert the strings to dates. This expression checks the values in the INVOICE_DATE port and converts each valid date to a date value. If the value is not a valid date, the Integration Service returns ERROR and skips the row.

This example returns a Date/Time value. Therefore, the output port for the expression needs to be Date/Time:

```
IIF( IS_DATE ( INVOICE_DATE, 'YYYY/MM/DD' ), TO_DATE( INVOICE_DATE ), ERROR('Not a valid date' ) )
```

```
INVOICE_DATE                    RETURN VALUE
NULL                            NULL
'180'                           'Not a valid date'
'04/01/98'                      'Not a valid date'
'1998/01/12'                    1998/01/12
'1998/11/21 00:00:13'           'Not a valid date'
'1998/02/31'                    'Not a valid date'
'John Smith'                    'Not a valid date'
```

IS_NUMBER

Availability:
Designer
Workflow Manager

Returns whether a string is a valid number. A valid number consists of the following parts:

- Optional space before the number
- Optional sign (+/-)
- One or more digits with an optional decimal point
- Optional scientific notation, such as the letter 'e' or 'E' (and the letter 'd' or 'D' on Windows) followed by an optional sign (+/-), followed by one or more digits
- Optional white space following the number

The following numbers are all valid:

```
' 100  '
'   +100'
'-100'
'-3.45e+32'
'+3.45E-32'
'+3.45d+32' (Windows only)
'+3.45D-32' (Windows only)
'.6804'
```

The output port for an IS_NUMBER expression must be a String or Numeric datatype.

You might use IS_NUMBER to test or filter data in a flat file before writing it to a target.

Syntax

```
IS_NUMBER( value )
```

Argument	Required/ Optional	Description
value	Required	Must be a String datatype. Passes the rows you want to evaluate. You can enter any valid transformation expression.

Return Value

TRUE (1) if the row is a valid number.

FALSE (0) if the row is not a valid number.

NULL if a value in the expression is NULL.

Examples

The following expression checks the ITEM_PRICE port for valid numbers:

```
IS_NUMBER( ITEM_PRICE )
```

ITEM_PRICE	RETURN VALUE
'123.00'	1 (True)
'-3.45e+3'	1 (True)
'-3.45D-3'	1 (True - Windows only)
'-3.45d-3'	0 (False - UNIX only)
'3.45E-'	0 (False) *Incomplete number*
' '	0 (False) *Consists entirely of blanks*
''	0 (False) *Empty string*
'+123abc'	0 (False)
' 123'	1 (True) *Leading white blanks*
'123 '	1 (True) *Trailing white blanks*
'ABC'	0 (False)
'-ABC'	0 (False)
NULL	NULL

Use IS_NUMBER to test data before using one of the numeric conversion functions, such as TO_FLOAT. For example, the following expression checks the values in the ITEM_PRICE port and converts each valid number to a double-precision floating point value. If the value is not a valid number, the Integration Service returns 0.00:

```
IIF( IS_NUMBER ( ITEM_PRICE ), TO_FLOAT( ITEM_PRICE ), 0.00 )
```

ITEM_PRICE	RETURN VALUE
'123.00'	123
'-3.45e+3'	-3450
'3.45E-3'	0.00345
' '	0.00 *Consists entirely of blanks*
''	0.00 *Empty string*

```
ITEM_PRICE              RETURN VALUE
'+123abc'               0.00
''  123ABC'             0.00
'ABC'                   0.00
'-ABC'                  0.00
NULL                    NULL
```

IS_SPACES

Availability:
Designer
Workflow Manager

Returns whether a string value consists entirely of spaces. A space is a blank space, a formfeed, a newline, a carriage return, a tab, or a vertical tab.

IS_SPACES evaluates an empty string as FALSE because there are no spaces. To test for an empty string, use LENGTH.

Syntax

```
IS_SPACES( value )
```

Argument	Required/ Optional	Description
value	Required	Must be a string datatype. Passes the rows you want to evaluate. You can enter any valid transformation expression.

Return Value

TRUE (1) if the row consists entirely of spaces.

FALSE (0) if the row contains data.

NULL if a value in the expression is NULL.

Example

The following expression checks the ITEM_NAME port for rows that consist entirely of spaces:

```
IS_SPACES( ITEM_NAME )
```

```
ITEM_NAME               RETURN VALUE
Flashlight              0 (False)
                        1 (True)
Regulator system        0 (False)
NULL                    NULL
''                      0 (FALSE) (Empty string does not contain spaces.)
```

Tip: Use IS_SPACES to avoid writing spaces to a character column in a target table. For example, if you have a transformation that writes customer names to a fixed length CHAR(5) column in a target table, you might want to write ‘00000’ instead of spaces. You would create an expression similar to the following:

```
IIF( IS_SPACES( CUST_NAMES ), '00000', CUST_NAMES )
```

LAST

Availability:
Designer

Returns the last row in the selected port. Optionally, you can apply a filter to limit the rows the Integration Service reads. You can nest only one other aggregate function within LAST.

Syntax

```
LAST( value [, filter_condition ] )
```

Argument	Required/ Optional	Description
value	Required	Any datatype except Binary. Passes the values for which you want to return the last row. You can enter any valid transformation expression.
filter_condition	Optional	Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression.

Return Value

Last row in a port.

NULL if all values passed to the function are NULL, or if no rows are selected (for example, the filter condition evaluates to FALSE or NULL for all rows).

Note: By default, the Integration Service treats null values as NULLs in aggregate functions. If you pass an entire port or group of null values, the function returns NULL. However, when you configure the Integration Service, you can choose how you want to handle null values in aggregate functions. You can treat null values as 0 in aggregate functions or as NULL.

Example

The following expression returns the last row in the ITEMS_NAME port with a price greater than $10.00:

```
LAST( ITEM_NAME, ITEM_PRICE > 10 )
```

ITEM_NAME	ITEM_PRICE
Flashlight	35.00
Navigation Compass	8.05
Regulator System	150.00
Flashlight	29.00
Depth/Pressure Gauge	88.00
Vest	31.00

RETURN VALUE: Vest

LAST_DAY

Availability:
Designer
Workflow Manager

Returns the date of the last day of the month for each date in a port.

Syntax

```
LAST_DAY( date )
```

Argument	Required/ Optional	Description
date	Required	Date/Time datatype. Passes the dates for which you want to return the last day of the month. You can enter any valid transformation expression that evaluates to a date.

Return Value

Date. The last day of the month for that date value you pass to this function.

NULL if a value in the selected port is NULL.

Null

If a value is NULL, LAST_DAY ignores the row. However, if all values passed from the port are NULL, LAST_DAY returns NULL.

Group By

LAST_DAY groups values based on group by ports you define in the transformation, returning one result for each group. If there is no group by port, LAST_DAY treats all rows as one group, returning one value.

Examples

The following expression returns the last day of the month for each date in the ORDER_DATE port:

```
LAST_DAY( ORDER_DATE )
```

```
ORDER_DATE                      RETURN VALUE
Apr 1 1998 12:00:00AM           Apr 30 1998 12:00:00AM
Jan 6 1998 12:00:00AM           Jan 31 1998 12:00:00AM
Feb 2 1996 12:00:00AM           Feb 29 1996 12:00:00AM   (Leap year)
NULL                            NULL
Jul 31 1998 12:00:00AM          Jul 31 1998 12:00:00AM
```

You can nest TO_DATE to convert string values to a date. TO_DATE always includes time information. If you pass a string that does not have a time value, the date returned will include the time 00:00:00.

The following example returns the last day of the month for each order date in the same format as the string:

```
LAST_DAY( TO_DATE( ORDER_DATE, 'DD-MON-YY' ))
```

```
ORDER_DATE          RETURN VALUE
'18-NOV-98'         Nov 30 1998 00:00:00
'28-APR-98'         Apr 30 1998 00:00:00
NULL                NULL
'18-FEB-96'         Feb 29 1996 00:00:00 (Leap year)
```

LEAST

Availability:
Designer
Workflow Manager

Returns the smallest value from a list of input values. By default, the match is case sensitive.

Syntax

```
LEAST( value1, [value2, ..., valueN,] CaseFlag )
```

Argument	Required/ Optional	Description
value	Required	Any datatype except Binary. Datatype must be compatible with other values. Value you want to compare against other values. You must enter at least one value argument. If the value is Numeric, and other input values are of other numeric datatypes, all values use the highest precision possible. For example, if some values are of the Integer datatype and others are of the Double datatype, the Integration Service converts the values to Double.
CaseFlag	Optional	Must be an integer. Determines whether the arguments in this function are case sensitive. You can enter any valid transformation expression. When *CaseFlag* is a number other than 0, the function is case sensitive. When *CaseFlag* is a null value or 0, the function is not case sensitive.

Return Value

value1 if it is the smallest of the input values, *value2* if it is the smallest of the input values, and so on.

NULL if any of the arguments is null.

Example

The following expression returns the smallest quantity of items ordered:

```
LEAST( QUANTITY1, QUANTITY2, QUANTITY3 )
```

QUANTITIY1	QUANTITY2	QUANTITY3	RETURN VALUE
150	756	27	27
			NULL
5000	97	17	17
120	1724	965	120

LENGTH

Availability:
Designer
Workflow Manager

Returns the number of characters in a string, including trailing blanks.

Syntax

```
LENGTH( string )
```

Argument	Required/ Optional	Description
string	Required	String datatype. The strings you want to evaluate. You can enter any valid transformation expression.

Return Value

Integer representing the length of the string.

NULL if a value passed to the function is NULL.

Example

The following expression returns the length of each customer name:

```
LENGTH( CUSTOMER_NAME )
```

CUSTOMER_NAME	RETURN VALUE
Bernice Davis	13
NULL	NULL
John Baer	9
Greg Brown	10

Tips for LENGTH

Use LENGTH to test for empty string conditions. If you want to find fields in which customer name is empty, use an expression such as:

```
IIF( LENGTH( CUSTOMER_NAME ) = 0, 'EMPTY STRING' )
```

To test for a null field, use ISNULL. To test for spaces, use IS_SPACES.

LN

Availability:
Designer
Workflow Manager

Returns the natural logarithm of a numeric value. For example, `LN(3)` returns `1.098612`. You usually use this function to analyze scientific data rather than business data.

This function is the reciprocal of the function EXP.

Syntax

```
LN( numeric_value )
```

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. It must be a positive number, greater than 0. Passes the values for which you want to calculate the natural logarithm. You can enter any valid transformation expression.

Return Value

Double value.

NULL if a value passed to the function is NULL.

Example

The following expression returns the natural logarithm for all values in the NUMBERS port:

```
LN( NUMBERS )
```

NUMBERS	RETURN VALUE
10	2.302585092994
125	4.828313737302
0.96	-0.04082199452026
NULL	NULL
-90	*Error. (The Integration Service does not write row.)*
0	*Error. (The Integration Service does not write row.)*

Note: The Integration Service displays an error and does not write the row when you pass a negative number or 0. The *numeric_value* must be a positive number greater than 0.

LOG

Availability:
Designer
Workflow Manager

Returns the logarithm of a numeric value. Most often, you use this function to analyze scientific data rather than business data.

Syntax

```
LOG( base, exponent )
```

Argument	Required/ Optional	Description
base	Required	The base of the logarithm. Must be a positive numeric value other than 0 or 1. Any valid transformation expression that evaluates to a positive number other than 0 or 1.
exponent	Required	The exponent of the logarithm. Must be a positive numeric value greater than 0. Any valid transformation expression that evaluates to a positive number greater than 0.

Return Value

Double value.

NULL if a value passed to the function is NULL.

Example

The following expression returns the logarithm for all values in the NUMBERS port:

```
LOG( BASE, EXPONENT )
```

```
BASE    EXPONENT    RETURN VALUE
15      1           0
.09     10          -0.956244644696599
NULL    18          NULL
35.78   NULL        NULL
-9      18          Error. (Integration Service does not write the row.)
0       5           Error. (Integration Service does not write the row.)
10      -2          Error. (Integration Service does not write the row.)
```

The Integration Service displays an error and does not write the row if you pass a negative number, 0, or 1 as a base value, or if you pass a negative value for the exponent.

LOOKUP

Availability:
Designer

Searches for a value in a lookup source column.

The LOOKUP function compares data in a lookup source to a value you specify. When the Integration Service finds the search value in the lookup table, it returns the value from a specified column in the same row in the lookup table.

When you create a session based on a mapping that uses the LOOKUP function, you must specify the database connections for $Source Connection Value and $Target Connection Value in the session properties. To validate a lookup function in an Expression transformation, verify that the lookup definition is in the mapping.

Note: This function is not supported in mapplets.

Using the Lookup Transformation or the LOOKUP Function

Use the Lookup *transformation* rather than the LOOKUP *function* to look up values in PowerCenter mappings. If you use the LOOKUP function in a mapping, you need to enable the lookup caching option for 3.5 compatibility in the session properties. This option exists expressly for PowerMart 3.5 users who want to continue using the LOOKUP function, rather than creating Lookup transformations. For more information, see “Lookup Transformation” in the *PowerCenter Transformation Guide*.

You can define multiple searches for one lookup table within a LOOKUP function. However, each search must find a matching value to return the lookup value.

Syntax

```
LOOKUP( result, search1, value1 [, search2, value2]... )
```

Argument	Required/ Optional	Description
result	Required	Any datatype except Binary. Must be an output port in the same lookup table as search. Specifies the return value if the search matches the value. Always preface this argument with the reference qualifier :TD.
search1	Required	Datatype that matches the *value1*. Must be an output port in the same lookup table as result. Specifies the values you want to match to value. Always preface this argument with the reference qualifier :TD.
value1	Required	Any datatype except Binary. Must match *search1* datatype. The values you want to search for in the lookup source column specified in *search1*. You can enter any valid transformation expression.

Return Value

Result if all searches find matching values. If the Integration Service finds matching values, it returns the result from the same row as the *search1* argument.

NULL if the search does not find any matching values.

Error if the search finds more than one matching value.

Example

The following expression searches the lookup source :TD.SALES for a specific item ID and price, and returns the item name if both searches find a match:

```
LOOKUP( :TD.SALES.ITEM_NAME, :TD.SALES.ITEM_ID, 10, :TD.SALES.PRICE, 15.99 )
```

ITEM_NAME	ITEM_ID	PRICE
Regulator	5	100.00
Flashlight	10	15.99
Halogen Flashlight	15	15.99
NULL	20	15.99

RETURN VALUE: Flashlight

Tips for LOOKUP

When you compare char and varchar values, the LOOKUP function returns a result only if the two rows match. This means that both the value and the length for each row must match. If the lookup source is a padded char value of a specified length and the lookup search is a varchar value, you need to use the RTRIM function to trim trailing blanks from the lookup source so that the values match the lookup search:

```
LOOKUP(:TD.ORDERS.PRICE, :TD.ORDERS.ITEM, RTRIM( ORDERS.ITEM, '   '))
```

Use the :TD reference qualifier in the *result* and *search* arguments of a LOOKUP function:

```
LOOKUP(:TD.ORDERS.ITEM, :TD.ORDERS.PRICE, ORDERS.PRICE, :TD.ORDERS.QTY, ORDERS.QTY)
```

LOWER

Availability:
Designer
Workflow Manager

Converts uppercase string characters to lowercase.

Syntax

```
LOWER( string )
```

Argument	Required/ Optional	Description
string	Required	Any string value. The argument passes the string values that you want to return as lowercase. You can enter any valid transformation expression that evaluates to a string.

Return Value

Lowercase character string. If the data contains multibyte characters, the return value depends on the code page and data movement mode of the Integration Service.

NULL if a value in the selected port is NULL.

Example

The following expression returns all first names to lowercase:

```
LOWER( FIRST_NAME )
```

FIRST_NAME	RETURN VALUE
antonia	antonia
NULL	NULL
THOMAS	thomas
PierRe	pierre
BERNICE	bernice

LPAD

Availability:
Designer
Workflow Manager

Adds a set of blanks or characters to the beginning of a string to set the string to a specified length.

Syntax

```
LPAD( first_string, length [,second_string] )
```

Argument	Required/ Optional	Description
first_string	Required	Can be a character string. The strings you want to change. You can enter any valid transformation expression.
length	Required	Must be a positive integer literal. This argument specifies the length you want each string to be.
second_string	Optional	Can be any string value. The characters you want to append to the left-side of the *first_string* values. You can enter any valid transformation expression. You can enter a specific string literal. However, enclose the characters you want to add to the beginning of the string within single quotation marks, as in 'abc'. This argument is case sensitive. If you omit the *second_string*, the function pads the beginning of the first string with blanks.

Return Value

String of the specified length.

NULL if a value passed to the function is NULL or if *length* is a negative number.

Examples

The following expression standardizes numbers to six digits by padding them with leading zeros.

```
LPAD( PART_NUM, 6, '0')
```

PART_NUM	RETURN VALUE
702	000702
1	000001
0553	000553
484834	484834

LPAD counts the length from left to right. If the first string is longer than the length, LPAD truncates the string from right to left. For example, LPAD(‘alphabetical’, 5, ‘x’) returns the string ‘alpha’.

If the second string is longer than the total characters needed to return the specified length, LPAD uses a portion of the second string:

```
LPAD( ITEM_NAME, 16, '*..*' )
```

ITEM_NAME	RETURN VALUE
Flashlight	*..**.Flashlight
Compass	*..**..**Compass
Regulator System	Regulator System
Safety Knife	*..*Safety Knife

LTRIM

Availability:
Designer
Workflow Manager

Removes blanks or characters from the beginning of a string. You can use LTRIM with IIF or DECODE in an Expression or Update Strategy transformation to avoid spaces in a target table.

If you do not specify a *trim_set* parameter in the expression:

- In UNICODE mode, LTRIM removes both single- and double-byte spaces from the beginning of a string.
- In ASCII mode, LTRIM removes only single-byte spaces.

If you use LTRIM to remove characters from a string, LTRIM compares the *trim_set* to each character in the *string* argument, character-by-character, starting with the left side of the string. If the character in the string matches any character in the *trim_set*, LTRIM removes it. LTRIM continues comparing and removing characters until it fails to find a matching character in the *trim_set*. Then it returns the string, which does not include matching characters.

Syntax

```
LTRIM( string [, trim_set] )
```

Arguments	Required/ Optional	Description
string	Required	Any string value. Passes the strings you want to modify. You can enter any valid transformation expression. Use operators to perform comparisons or concatenate strings before removing characters from the beginning of a string.
trim_set	Optional	Any string value. Passes the characters you want to remove from the beginning of the first string. You can enter any valid transformation expression. You can also enter a character string. However, you must enclose the characters you want to remove from the beginning of the string within single quotation marks, for example, 'abc'. If you omit the second string, the function removes any blanks from the beginning of the string. LTRIM is case sensitive. For example, if you want to remove the 'A' character from the string 'Alfredo', you would enter 'A', not 'a'.

Return Value

String. The string values with the specified characters in the *trim_set* argument removed.

NULL if a value passed to the function is NULL. If the *trim_set* is NULL, the function returns NULL.

Example

The following expression removes the characters ‘S’ and ‘.’ from the strings in the LAST_NAME port:

```
LTRIM( LAST_NAME, 'S.')
```

LAST_NAME	RETURN VALUE
Nelson	Nelson
Osborne	Osborne
NULL	NULL
S. MacDonald	MacDonald
Sawyer	awyer
H. Bender	H. Bender
Steadman	teadman

LTRIM removes ‘S.’ from S. MacDonald and the ‘S’ from both Sawyer and Steadman, but not the period from H. Bender. This is because LTRIM searches, character-by-character, for the set of characters you specify in the *trim_set* argument. If the first character in the string matches the first character in the *trim_set*, LTRIM removes it. Then LTRIM looks at the second character in the string. If it matches the second character in the *trim_set*, LTRIM removes it, and so on. When the first character in the string does not match the corresponding character in the *trim_set*, LTRIM returns the string and evaluates the next row.

In the example of H. Bender, H does not match either character in the *trim_set* argument, so LTRIM returns the string in the LAST_NAME port and moves to the next row.

Tips for LTRIM

Use RTRIM and LTRIM with || or CONCAT to remove leading and trailing blanks after you concatenate two strings.

You can also remove multiple sets of characters by nesting LTRIM. For example, if you want to remove leading blanks and the character 'T' from a column of names, you might create an expression similar to the following:

```
LTRIM( LTRIM( NAMES ), 'T' )
```

MAKE_DATE_TIME

Availability:
Designer
Workflow Manager

Returns the date and time based on the input values.

Syntax

```
MAKE_DATE_TIME( year, month, day, hour, minute, second, nanosecond )
```

Argument	Required/ Optional	Description
year	Required	Numeric datatype. Positive 4-digit integer. If you pass this function a 2-digit year, the Integration Service returns “00” as the first two digits of the year.
month	Required	Numeric datatype. Positive integer between 1 and 12 (January=1 and December=12).
day	Required	Numeric datatype. Positive integer between 1 and 31 (except for the months that have less than 31 days: February, April, June, September, and November).
hour	Optional	Numeric datatype. Positive integer between 0 and 24 (where 0=12AM, 12=12PM, and 24 =12AM).
minute	Optional	Numeric datatype. Positive integer between 0 and 59.
second	Optional	Numeric datatype. Positive integer between 0 and 59.
nanosecond	Optional	Numeric datatype. Positive integer between 0 and 999,999,999.

Return Value

Date as MM/DD/YYYY HH24:MI:SS. Returns a null value if you do not pass the function a year, month, or day.

Example

The following expression creates a date and time from the input ports:

```
MAKE_DATE_TIME( SALE_YEAR, SALE_MONTH, SALE_DAY, SALE_HOUR, SALE_MIN, SALE_SEC )
```

SALE_YR	SALE_MTH	SALE_DAY	SALE_HR	SALE_MIN	SALE_SEC	RETURN VALUE
2002	10	27	8	36	22	10/27/2002 08:36:22
2000	6	15	15	17		06/15/2000 15:17:00
2003	1	3		22	45	01/03/2003 00:22:45
04	3	30	12	5	10	03/30/0004 12:05:10
99	12	12	5		16	12/12/0099 05:00:16

MAX (Dates)

Availability:
Designer

Returns the latest date found within a port or group. You can apply a filter to limit the rows in the search. You can nest only one other aggregate function within MAX.

You can also use MAX to return the largest numeric value or the highest string value in a port or group.

Syntax

```
MAX( date [, filter_condition] )
```

Argument	Required/ Optional	Description
date	Required	Date/Time datatype. Passes the date for which you want to return a maximum date. You can enter any valid transformation expression.
filter_condition	Optional	Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression.

Return Value

Date.

NULL if all values passed to the function are NULL, or if no rows are selected (for example, the filter condition evaluates to FALSE or NULL for all rows).

Example

You can return the maximum date for a port or group. The following expression returns the maximum order date for flashlights:

```
MAX( ORDERDATE, ITEM_NAME='Flashlight' )
```

ITEM_NAME	ORDER_DATE
Flashlight	Apr 20 1998
Regulator System	May 15 1998
Flashlight	Sep 21 1998
Diving Hood	Aug 18 1998
Flashlight	NULL

MAX (Numbers)

Availability:
Designer

Returns the maximum numeric value found within a port or group. You can apply a filter to limit the rows in the search. You can nest only one other aggregate function within MAX. You can also use MAX to return the latest date or the highest string value in a port or group.

Syntax

```
MAX( numeric_value [, filter_condition] )
```

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. Passes the numeric values for which you want to return a maximum numeric value. You can enter any valid transformation expression.
filter_condition	Optional	Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression.

Return Value

Numeric value.

NULL if all values passed to the function are NULL or if no rows are selected (for example, the filter condition evaluates to FALSE or NULL for all rows).

If the return value is decimal with precision greater than 15, you can enable high precision in the session properties to ensure decimal precision up to 28 digits.

Nulls

If a value is NULL, MAX ignores it. However, if all values passed from the port are NULL, MAX returns NULL.

Note: By default, the Integration Service treats null values as NULLs in aggregate functions. If you pass an entire port or group of null values, the function returns NULL. However, when you configure the Integration Service, you can choose how you want to handle null values in aggregate functions. You can treat null values as 0 in aggregate functions or as NULL.

Group By

MAX groups values based on group by ports you define in the transformation, returning one result for each group.

If there is no group by port, MAX treats all rows as one group, returning one value.

Example

The first expression returns the maximum price for flashlights:

```
MAX( PRICE, ITEM_NAME='Flashlight' )
```

ITEM_NAME	PRICE
Flashlight	10.00
Regulator System	360.00
Flashlight	55.00
Diving Hood	79.00
Halogen Flashlight	162.00
Flashlight	85.00

```
ITEM_NAME                                                    PRICE
Flashlight                                                   NULL
RETURN VALUE:  85.00
```

MAX (String)

Availability:
Designer

Returns the highest string value found within a port or group. You can apply a filter to limit the rows in the search. You can nest only one other aggregate function within MAX.

Note: The MAX function uses the same sort order that the Sorter transformation uses. However, the MAX function is case sensitive, and the Sorter transformation may not be case sensitive.

You can also use MAX to return the latest date or the largest numeric value in a port or group.

Syntax

```
MAX( string [, filter_condition] )
```

Argument	Required/ Optional	Description
string	Required	String datatype. Passes the string values for which you want to return a maximum string value. You can enter any valid transformation expression.
filter_condition	Optional	Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression.

Return Value

String.

NULL if all values passed to the function are NULL, or if no rows are selected (for example, the filter condition evaluates to FALSE or NULL for all rows).

Nulls

If a value is NULL, MAX ignores it. However, if all values passed from the port are NULL, MAX returns NULL.

Note: By default, the Integration Service treats null values as NULLs in aggregate functions. If you pass an entire port or group of null values, the function returns NULL. However, when you configure the Integration Service, you can choose how you want to handle null values in aggregate functions. You can treat null values as 0 in aggregate functions or as NULL.

Group By

MAX groups values based on group by ports you define in the transformation, returning one result for each group.

If there is no group by port, MAX treats all rows as one group, returning one value.

Example

The following expression returns the maximum item name for manufacturer ID 104:

```
MAX( ITEM_NAME, MANUFACTURER_ID='104' )
```

MANUFACTURER_ID	ITEM_NAME
101	First Stage Regulator
102	Electronic Console
104	Flashlight
104	Battery (9 volt)
104	Rope (20 ft)
104	60.6 cu ft Tank
107	75.4 cu ft Tank
108	Wristband Thermometer

RETURN VALUE: Rope (20 ft)

MD5

Availability:
Designer
Workflow Manager

Calculates the checksum of the input value. The function uses Message-Digest algorithm 5 (MD5). MD5 is a one-way cryptographic hash function with a 128-bit hash value. You can conclude that input values are different when the checksums of the input values are different. Use MD5 to verify data integrity.

Syntax

```
MD5( value )
```

Argument	Required/ Optional	Description
value	Required	String or Binary datatype. Value for which you want to calculate checksum. The case of the input value affects the return value. For example, MD5(informatica) and MD5(Informatica) return different values.

Return Value

Unique 32-character string of hexadecimal digits 0-9 and a-f.

NULL if the input is a null value.

Example

You want to write changed data to a database. Use MD5 to generate checksum values for rows of data you read from a source. When you run a session, compare the previously generated checksum values against the new checksum values. Then, write the rows with updated checksum values to the target. You can conclude that an updated checksum value indicates that the data has changed.

Tip

You can use the return value as a hash key.

MEDIAN

Availability:
Designer

Returns the median of all values in a selected port.

If there is an even number of values in the port, the median is the average of the middle two values when all values are placed ordinally on a number line. If there is an odd number of values in the port, the median is the middle number.

You can nest only one other aggregate function within MEDIAN, and the nested function must return a Numeric datatype.

The Integration Service reads all rows of data to perform the median calculation. The process of reading rows of data to perform the calculation may affect performance. Optionally, you can apply a filter to limit the rows you read to calculate the median.

Syntax

```
MEDIAN( numeric_value [, filter_condition ] )
```

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. Passes the values for which you want to calculate a median. You can enter any valid transformation expression.
filter_condition	Optional	Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression.

Return Value

Numeric value.

NULL if all values passed to the function are NULL, or if no rows are selected. For example, the filter condition evaluates to FALSE or NULL for all rows.

Note: If the return value is decimal with precision greater than 15, you can enable high precision in the session properties to ensure decimal precision up to 28 digits.

Nulls

If a value is NULL, MEDIAN ignores the row. However, if all values passed from the port are NULL, MEDIAN returns NULL.

Note: By default, the Integration Service treats null values as NULLs in aggregate functions. If you pass an entire port or group of null values, the function returns NULL. However, when you configure the Integration Service, you can choose how you want to handle null values in aggregate functions. You can treat null values as 0 in aggregate functions or as NULL.

Group By

MEDIAN groups values based on group by ports you define in the transformation, returning one result for each group.

If there is no group by port, MEDIAN treats all rows as one group, returning one value.

Example

To calculate the median salary for all departments, you create an Aggregator transformation grouped by departments with a port specifying the following expression:

```
MEDIAN( SALARY )
```

The following expression returns the median value for orders of stabilizing vests:

```
MEDIAN( SALES, ITEM = 'Stabilizing Vest' )
```

ITEM	SALES
Flashlight	85
Stabilizing Vest	504
Stabilizing Vest	36
Safety Knife	5
Medium Titanium Knife	150
Tank	NULL
Stabilizing Vest	441
Chisel Point Knife	60
Stabilizing Vest	NULL
Stabilizing Vest	1044
Wrist Band Thermometer	110

RETURN VALUE: 472.5

METAPHONE

Availability:
Designer
Workflow Manager

Encodes string values. You can specify the length of the string that you want to encode.

METAPHONE encodes characters of the English language alphabet (A-Z). It encodes both uppercase and lowercase letters in uppercase.

METAPHONE encodes characters according to the following list of rules:

- Skips vowels (A, E, I, O, and U) unless one of them is the first character of the input string. METAPHONE(‘CAR’) returns ‘KR’ and METAPHONE(‘AAR’) returns ‘AR’.
- Uses special encoding guidelines.

The following table lists the METAPHONE encoding guidelines:

Input	Returns	Condition	Example
B	- n/a	- when it follows M	- METAPHONE (‘Lamb’) returns LM.
	- B	- in all other cases	- METAPHONE (‘Box’) returns BKS.

Input	Returns	Condition	Example
C	- X	- when followed by IA or H	- METAPHONE (‘Facial’) returns FXL.
	- S	- when followed by I, E, or Y	- METAPHONE (‘Fence’) returns FNS.
	- n/a	- when it follows S, and is followed by I, E, or Y -	- METAPHONE (‘Scene’) returns SN.
	- K	- in all other cases	- METAPHONE (‘Cool’) returns KL.
D	- J	- when followed by GE, GY, or GI	- METAPHONE (‘Dodge’) returns TJ.
	- T	- in all other cases	- METAPHONE (‘David’) returns TFT.
F	- F	- in all cases	- METAPHONE (‘FOX’) returns FKS.
G	- F	- when followed by H and the first character in the input string is not B, D, or H	- METAPHONE (‘Tough’) returns TF.
	- n/a	- when followed by H and the first character in the input string is B, D, or H	- METAPHONE (‘Hugh’) returns HF.
	- J	- when followed by I, E or Y and does not repeat	- METAPHONE (‘Magic’) returns MJK.
	- K	- in all other cases	- METAPHONE(‘GUN’) returns KN.
H	- H	- when it does not follow C, G, P, S, or T and is followed by A, E, I, or U	- METAPHONE (‘DHAT’) returns THT.
	- n/a	- in all other cases	- METAPHONE (‘Chain’) returns XN.
J	- J	- in all cases	- METAPHONE (‘Jen’) returns JN.
K	- n/a - K	- when it follows C - in all other cases	- METAPHONE (‘Ckim’) returns KM. - METAPHONE (’Kim’) returns KM.
L	- L	- in all cases	- METAPHONE (‘Laura’) returns LR.
M	- M	- in all cases	- METAPHONE (‘Maggi’) returns MK.
N	- N	- in all cases	- METAPHONE (‘Nancy’) returns NNS.
P	- F	- when followed by H	- METAPHONE (‘Phone’) returns FN.
	- P	- in all other cases	- METAPHONE (‘Pip’) returns PP.
Q	- K	- in all cases	- METAPHONE (‘Queen’) returns KN.
R	- R	- in all cases	- METAPHONE (‘Ray’) returns R.
S	- X	- when followed by H, IO, IA, or CHW	- METAPHONE (‘Cash’) returns KX.
	- S	- in all other cases	- METAPHONE (‘Sing’) returns SNK.

Input	Returns	Condition	Example
T	- X	- when followed by IA or IO	- METAPHONE (‘Patio’) returns PX.
	- 0[1]	- when followed by H	- METAPHONE (‘Thor’) returns 0R.
	- n/a	- when followed by CH	- METAPHONE (‘Glitch’) returns KLTX.
	- T	- in all other cases	- METAPHINE (‘Tim’) returns TM.
V	- F	- in all cases	- METAPHONE (‘Vin’) returns FN.
W	- W	- when followed by A, E, I, O, or U	- METAPHONE (‘Wang’) returns WNK.
	- n/a	- in all other cases	- METAPHONE (‘When’) returns HN.
X	- KS	- in all cases	- METAPHONE (‘Six’) returns SKS.
Y	- Y	- when followed by A, E, I, O, or U	- METAPHONE (‘Yang’) returns YNK.
	- n/a	- in all other cases	- METAPHONE (‘Bobby’) returns BB.
Z	- S	- in all cases	- METAPHONE (‘Zack’) returns SK.
1. The integer 0.			

- Skips the initial character and encodes the remaining string if the first two characters of the input string have one of the following values:
  - **KN.** For example, METAPHONE(‘KNOT’) returns ‘NT’.
  - **GN.** For example, METAPHONE(‘GNOB’) returns ‘NB’.
  - **PN.** For example, METAPHONE(‘PNRX’) returns ‘NRKS’.
  - **AE.** For example, METAPHONE(‘AERL’) returns ‘ERL’.
- If a character other than “C” occurs more than once in the input string, encodes the first occurrence only. For example, METAPHONE(‘BBOX’) returns ‘BKS’ and METAPHONE(‘CCOX’) returns ‘KKKS’.

Syntax

```
METAPHONE( string [,length] )
```

Argument	Required/ Optional	Description
string	Required	Must be a character string. Passes the value you want to encode. The first character must be a character in the English language alphabet (A-Z). You can enter any valid transformation expression. Skips any non-alphabetic character in *string*.
length	Optional	Must be an integer greater than 0. Specifies the number of characters in *string* that you want to encode. You can enter any valid transformation expression. When *length* is 0 or a value greater than the length of *string*, encodes the entire input string. Default is 0.

Return Value

String.

NULL if one of the following conditions is true:

- All values passed to the function are NULL.
- No character in *string* is a letter of the English alphabet.
- *string* is empty.

Examples

The following expression encodes the first two characters in EMPLOYEE_NAME port to a string:

```
METAPHONE( EMPLOYEE_NAME, 2 )
```

Employee_Name	Return Value
John	JH
*@#$	NULL
P$%%oc&&KMNL	PK

The following expression encodes the first four characters in EMPLOYEE_NAME port to a string:

```
METAPHONE( EMPLOYEE_NAME, 4 )
```

Employee_Name	Return Value
John	JHN
1ABC	ABK
*@#$	NULL
P$%%oc&&KMNL	PKKM

MIN (Dates)

Availability:
Designer

Returns the earliest date found in a port or group. You can apply a filter to limit the rows in the search. You can nest only one other aggregate function within MIN, and the nested function must return a date datatype.

You can also use MIN to return the smallest numeric value or the lowest string value in a port or group.

Syntax

```
MIN( date [, filter_condition] )
```

Argument	Required/ Optional	Description
date	Required	Date/Time datatype. Passes the values for which you want to return minimum value. You can enter any valid transformation expression.
filter_condition	Optional	Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression.

Return Value

Date if the *value* argument is a date.

NULL if all values passed to the function are NULL, or if no rows are selected (for example, the filter condition evaluates to FALSE or NULL for all rows).

Nulls

If a single value is NULL, MIN ignores it. However, if all values passed from the port are NULL, MIN returns NULL.

Group By

MIN groups values based on group by ports you define in the transformation, returning one result for each group.

If there is no group by port, MIN treats all rows as one group, returning one value.

Example

The following expression returns the oldest order date for flashlights:

```
MIN( ORDER_DATE, ITEM_NAME='Flashlight' )
```

ITEM_NAME	ORDER_DATE
Flashlight	Apr 20 1998
Regulator System	May 15 1998
Flashlight	Sep 21 1998
Diving Hood	Aug 18 1998
Halogen Flashlight	Feb 1 1998
Flashlight	Oct 10 1998
Flashlight	NULL

RETURN VALUE: Feb 1 1998

MIN (Numbers)

Availability:
Designer

Returns the smallest numeric value found in a port or group. You can apply a filter to limit the rows in the search. You can nest only one other aggregate function within MIN, and the nested function must return a numeric datatype.

You can also use MIN to return the latest date or the lowest string value in a port or group.

Syntax

```
MIN( numeric_value [, filter_condition] )
```

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatypes. Passes the values for which you want to return minimum value. You can enter any valid transformation expression.
filter_condition	Optional	Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression.

Return Value

Numeric value.

NULL if all values passed to the function are NULL, or if no rows are selected (for example, the filter condition evaluates to FALSE or NULL for all rows).

Note: If the return value is a decimal with precision greater than 15, you can enable high precision in the session properties to ensure decimal precision up to 28 digits.

Nulls

If a single value is NULL, MIN ignores it. However, if all values passed from the port are NULL, MIN returns NULL.

Note: By default, the Integration Service treats null values as NULLS in aggregate functions. If you pass an entire port or group of null values, the function returns NULL. However, when you configure the Integration Service, you can choose how you want to handle null values in aggregate functions. You can treat null values as 0 in aggregate functions or as NULL.

Group By

MIN groups values based on group by ports you define in the transformation, returning one result for each group.

If there is no group by port, MIN treats all rows as one group, returning one value.

Example

The following expression returns the minimum price for flashlights:

```
MIN ( PRICE, ITEM_NAME='Flashlight' )
```

ITEM_NAME	PRICE
Flashlight	10.00
Regulator System	360.00
Flashlight	55.00
Diving Hood	79.00
Halogen Flashlight	162.00
Flashlight	85.00
Flashlight	NULL

RETURN VALUE: 10.00

MIN (String)

Availability:
Designer

Returns the lowest string value found in a port or group. You can apply a filter to limit the rows in the search. You can nest only one other aggregate function within MIN, and the nested function must return a string datatype.

Note: The MIN function uses the same sort order that the Sorter transformation uses. However, the MIN function is case sensitive, but the Sorter transformation may not be case sensitive.

You can also use MIN to return the latest date or the minimum numeric value in a port or group.

Syntax

```
MIN( string [, filter_condition] )
```

Argument	Required/ Optional	Description
string	Required	String datatype. Passes the values for which you want to return minimum value. You can enter any valid transformation expression.
filter_condition	Optional	Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression.

Return Value

String value.

NULL if all values passed to the function are NULL, or if no rows are selected (for example, the filter condition evaluates to FALSE or NULL for all rows).

Nulls

If a single value is NULL, MIN ignores it. However, if all values passed from the port are NULL, MIN returns NULL.

Note: By default, the Integration Service treats null values as NULLS in aggregate functions. If you pass an entire port or group of null values, the function returns NULL. However, when you configure the Integration Service, you can choose how you want to handle null values in aggregate functions. You can treat null values as 0 in aggregate functions or as NULL.

Group By

MIN groups values based on group by ports you define in the transformation, returning one result for each group.

If there is no group by port, MIN treats all rows as one group, returning one value.

Example

The following expression returns the minimum item name for manufacturer ID 104:

```
MIN ( ITEM_NAME, MANUFACTURER_ID='104' )
```

MANUFACTURER_ID	ITEM_NAME
101	First Stage Regulator
102	Electronic Console
104	Flashlight
104	Battery (9 volt)
104	Rope (20 ft)
104	60.6 cu ft Tank
107	75.4 cu ft Tank
108	Wristband Thermometer

RETURN VALUE: 60.6 cu ft Tank

MOD

Availability:
Designer
Workflow Manager

Returns the remainder of a division calculation. For example, `MOD(8,5)` returns 3.

Syntax

```
MOD( numeric_value, divisor )
```

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. The values you want to divide. You can enter any valid transformation expression.
divisor	Required	The numeric value you want to divide by. The divisor cannot be 0.

Return Value

Numeric value of the datatype you pass to the function. The remainder of the numeric value divided by the divisor.

NULL if a value passed to the function is NULL.

Examples

The following expression returns the modulus of the values in the PRICE port divided by the values in the QTY port:

```
MOD( PRICE, QTY )
```

PRICE	QTY	RETURN VALUE
10.00	2	0
12.00	5	2
9.00	2	1
15.00	3	0
NULL	3	NULL
20.00	NULL	NULL
25.00	0	*Error. Integration Service does not write row.*

The last row (25, 0) produced an error because you cannot divide by 0. To avoid dividing by 0, you can create an expression similar to the following, which returns the modulus of Price divided by Quantity only if the quantity is not 0. If the quantity is 0, the function returns NULL:

```
MOD( PRICE, IIF( QTY = 0, NULL, QTY ))
```

PRICE	QTY	RETURN VALUE
10.00	2	0
12.00	5	2
9.00	2	1
15.00	3	0
NULL	3	NULL
20.00	NULL	NULL
25.00	0	NULL

The last row (25, 0) produced a NULL rather than an error because the IIF function replaces NULL with the 0 in the QTY port.

MOVINGAVG

Availability:
Designer

Returns the average (row-by-row) of a specified set of rows. Optionally, you can apply a condition to filter rows before calculating the moving average.

Syntax

```
MOVINGAVG( numeric_value, rowset [, filter_condition] )
```

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. The values for which you want to calculate a moving average. You can enter any valid transformation expression.
rowset	Required	Must be a positive integer literal greater than 0. Defines the row set for which you want to calculate the moving average. For example, if you want to calculate a moving average for a column of data, five rows at a time, you might write an expression such as: `MOVINGAVG(SALES, 5)`.
filter_condition	Optional	Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression.

Return Value

Numeric value.

NULL if all values passed to the function are NULL or if no rows are selected (for example, the filter condition evaluates to FALSE or NULL for all rows).

Note: If the return value is a decimal with precision greater than 15, you can enable high precision in the session properties to ensure decimal precision up to 28 digits.

Nulls

MOVINGAVG ignores null values when calculating the moving average. However, if all values are NULL, the function returns NULL.

Example

The following expression returns the average order for a Stabilizing Vest, based on the first five rows in the Sales port, and thereafter, returns the average for the last five rows read:

```
MOVINGAVG( SALES, 5 )
```

ROW_NO	SALES	RETURN VALUE
1	600	NULL
2	504	NULL
3	36	NULL
4	100	NULL
5	550	358
6	39	245.8
7	490	243

The function returns the average for a set of five rows: 358 based on rows 1 through 5, 245.8 based on rows 2 through 6, and 243 based on rows 3 through 7.

MOVINGSUM

Availability:
Designer

Returns the sum (row-by-row) of a specified set of rows.

Optionally, you can apply a condition to filter rows before calculating the moving sum.

Syntax

```
MOVINGSUM( numeric_value, rowset [, filter_condition] )
```

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. The values for which you want to calculate a moving sum. You can enter any valid transformation expression.
rowset	Required	Must be a positive integer literal greater than 0. Defines the rowset for which you want to calculate the moving sum. For example, if you want to calculate a moving sum for a column of data, five rows at a time, you might write an expression such as: `MOVINGSUM( SALES, 5 )`
filter_condition	Optional	Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression.

Return Value

Numeric value.

NULL if all values passed to the function are NULL, or if the function does not select any rows (for example, the filter condition evaluates to FALSE or NULL for all rows).

Note: If the return value is a Decimal with precision greater than 15, you can enable high precision in the session properties to ensure decimal precision up to 28 digits.

Nulls

MOVINGSUM ignores null values when calculating the moving sum. However, if all values are NULL, the function returns NULL.

Example

The following expression returns the sum of orders for a Stabilizing Vest, based on the first five rows in the Sales port, and thereafter, returns the average for the last five rows read:

```
MOVINGSUM( SALES, 5 )
```

ROW_NO	SALES	RETURN VALUE
1	600	NULL
2	504	NULL
3	36	NULL
4	100	NULL
5	550	1790
6	39	1229
7	490	1215

The function returns the sum for a set of five rows: 1790 based on rows 1 through 5, 1229 based on rows 2 through 6, and 1215 based on rows 3 through 7.

NPER

Availability:
Designer
Workflow Manager

Returns the number of periods for an investment based on a constant interest rate and periodic, constant payments.

Syntax

```
NPER( rate, present value, payment [, future value, type] )
```

Argument	Required/ Optional	Description
rate	Required	Numeric. Interest rate earned in each period. Expressed as a decimal number. Divide the rate by 100 to express it as a decimal number. Must be greater than or equal to 0.
present value	Required	Numeric. Lump-sum amount a series of future payments is worth.
payment	Required	Numeric. Payment amount due per period. Must be a negative number.
future value	Optional	Numeric. Cash balance you want to attain after the last payment is made. If you omit this value, NPER uses 0.
type	Optional	Boolean. Timing of the payment. Enter 1 if payment is at the beginning of period. Enter 0 if payment is at the end of period. Default is 0. If you enter a value other than 0 or 1, the Integration Service treats the value as 1.

Return Value

Numeric.

Example

The present value of an investment is $2,000. Each payment is $500 and the future value of the investment is $20,000. The following expression returns 9 as the number of periods for which you need to make the payments:

```
NPER( 0.01, -2000, -500, 20000, TRUE )
```

Notes

To calculate interest rate earned in each period, divide the annual rate by the number of payments made in an year. For example, if you make monthly payments at an annual interest rate of 15 percent, the value of the Rate argument is 15% divided by 12. If you make annual payments, the value of the Rate argument is 15%.

The payment value and present value are negative because these are amounts that you pay.

PERCENTILE

Availability:
Designer

Calculates the value that falls at a given percentile in a group of numbers. You can nest only one other aggregate function within PERCENTILE, and the nested function must return a Numeric datatype.

The Integration Service reads all rows of data to perform the percentile calculation. The process of reading rows to perform the calculation may affect performance. Optionally, you can apply a filter to limit the rows you read to calculate the percentile.

Syntax

```
PERCENTILE( numeric_value, percentile [, filter_condition ] )
```

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. Passes the values for which you want to calculate a percentile. You can enter any valid transformation expression.
percentile	Required	Integer between 0 and 100, inclusive. Passes the percentile you want to calculate. You can enter any valid transformation expression. If you pass a number outside the 0 to 100 range, the Integration Service displays an error and does not write the row.
filter_condition	Optional	Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression.

Return Value

Numeric value.

NULL if all values passed to the function are NULL, or if no rows are selected (for example, the filter condition evaluates to FALSE or NULL for all rows).

Note: If the return value is decimal with precision greater than 15, you can enable high precision in the session properties to ensure decimal precision up to 28 digits.

Nulls

If a value is NULL, PERCENTILE ignores the row. However, if all values in a group are NULL, PERCENTILE returns NULL.

Note: By default, the Integration Service treats null values as NULLs in aggregate functions. If you pass an entire port or group of null values, the function returns NULL. However, when you configure the Integration Service, you can choose how you want to handle null values in aggregate functions. You can treat null values as 0 in aggregate functions or as NULL.

Group By

PERCENTILE groups values based on group by ports you define in the transformation, returning one result for each group.

If there is no group by port, PERCENTILE treats all rows as one group, returning one value.

Example

The Integration Service calculates a percentile using the following logic:

$$i = \frac{(x + 1) \times \text{percentile}}{100}$$

Use the following guidelines for this equation:

- *x* is the number of elements in the group of values for which you are calculating a percentile.
- If *i* < 1, PERCENTILE returns the value of the first element in the list.
- If *i* is an integer value, PERCENTILE returns the value of the *i*th element in the list.
- Otherwise PERCENTILE returns the value of *n*:

$$n = [\lfloor i \rfloor \text{th?element} \times (\lceil i \rceil - i)] + [\lceil i \rceil \text{th?element} \times (i - \lfloor i \rfloor)]$$

The following expression returns the salary that falls at the 75th percentile of salaries greater than $50,000:

```
PERCENTILE( SALARY, 75, SALARY > 50000 )
```

```
SALARY
125000.0
27900.0
100000.0
NULL
55000.0
9000.0
85000.0
86000.0
48000.0
99000.0
RETURN VALUE:  106250.0
```

PMT

Availability:
Designer
Workflow Manager

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

Syntax

```
PMT( rate, terms, present value[, future value, type] )
```

Argument	Required/ Optional	Description
rate	Required	Numeric. Interest rate of the loan for each period. Expressed as a decimal number. Divide the rate by 100 to express it as a decimal number. Must be greater than or equal to 0.
terms	Required	Numeric. Number of periods or payments. Must be greater than 0.
present value	Required	Numeric. Principle for the loan.

Argument	Required/ Optional	Description
future value	Optional	Numeric. Cash balance you want to attain after the last payment. If you omit this value, PMT uses 0.
type	Optional	Boolean. Timing of the payment. Enter 1 if the payment is at the beginning of period. Enter 0 if the payment is at the end of period. Default is 0. If you enter a value other than 0 or 1, the Integration Service treats the value as 1.

Return Value

Numeric.

Example

The following expression returns -2111.64 as the monthly payment amount of a loan:

```
PMT( 0.01, 10, 20000 )
```

Notes

To calculate interest rate earned in each period, divide the annual rate by the number of payments made in a year. For example, if you make monthly payments at an annual interest rate of 15%, the rate is 15%/12. If you make annual payments, the rate is 15%.

The payment value is negative because these are amounts that you pay.

POWER

Availability:
Designer
Workflow Manager

Returns a value raised to the exponent you pass to the function.

Syntax

```
POWER( base, exponent )
```

Argument	Required/ Optional	Description
base	Required	Numeric value. This argument is the base value. You can enter any valid transformation expression. If the base value is negative, the exponent must be an integer.
exponent	Required	Numeric value. This argument is the exponent value. You can enter any valid transformation expression. If the base value is negative, the exponent must be an integer. In this case, the function rounds any decimal values to the nearest integer before returning a value.

Return Value

Double value.

NULL if you pass a null value to the function.

Example

The following expression returns the values in the Numbers port raised to the values in the Exponent port:

```
POWER( NUMBERS, EXPONENT )
```

NUMBERS	EXPONENT	RETURN VALUE
10.0	2.0	100
3.5	6.0	1838.265625
3.5	5.5	982.594307804838
NULL	2.0	NULL
10.0	NULL	NULL
-3.0	-6.0	0.00137174211248285
3.0	-6.0	0.00137174211248285
-3.0	6.0	729.0
-3.0	5.5	729.0

The value -3.0 raised to 6 returns the same results as -3.0 raised to 5.5. If the base is negative, the exponent must be an integer. Otherwise, the Integration Service rounds the exponent to the nearest integer value.

PV

Availability:
Designer
Workflow Manager

Returns the present value of an investment.

Syntax

```
PV( rate, terms, payment [, future value, type] )
```

Argument	Required/ Optional	Description
rate	Required	Numeric. Interest rate earned in each period. Expresses as a decimal number. Divide the rate by 100 to express it as a decimal number. Must be greater that or equal to 0.
terms	Required	Numeric. Number of period or payments. Must be greater than 0.
payments	Required	Numeric. Payment amount due per period. Must be a negative number.
future value	Optional	Numeric. Cash balance after the last payment. If you omit this value, PV uses 0.
types	Optional	Boolean. Timing of the payment. Enter 1 if payment is at the beginning of period. Enter 0 if the payment is at the end of period. Default is 0. If you enter a value other than 0 or 1, the Integration Service treats the value as 1.

Return Value

Numeric.

Example

The following expression returns 12,524.43 as the amount you must deposit in the account today to have a future value of $20,000 in one year if you also deposit $500 at the beginning of each period:

```
PV( 0.0075, 12, -500, 20000, TRUE )
```

RAND

Availability:
Designer
Workflow Manager

Returns a random number between 0 and 1. This is useful for probability scenarios.

Syntax

```
RAND( seed )
```

Argument	Required/ Optional	Description
seed	Optional	Numeric. Starting value for the Integration Service to generate the random number. Value must be a constant. If you do not enter a seed, the Integration Service uses the current system time to derive the numbers of seconds since January 1, 1971. It uses this value as the seed.

Return Value

Numeric.

For the same seed, the Integration Service generates the same sequence of numbers.

Example

The following expression may return a value of 0.417022004702574:

```
RAND (1)
```

RATE

Availability:
Designer
Workflow Manager

Returns the interest rate earned per period by a security.

Syntax

```
RATE( terms, payment, present value[, future value, type] )
```

Argument	Required/ Optional	Description
terms	Required	Numeric. Number of periods or payments. Must be greater than 0.
payments	Required	Numeric. Payment amount due per period. Must be a negative number.
present value	Required	Numeric. Lump-sum amount that a series of future payments is worth now.
future value	Optional	Numeric. Cash balance you want to attain after the last payment. For example, the future value of a loan is 0. If you omit this argument, RATE uses 0.
types	Optional	Boolean. Timing of the payment. Enter 1 if payment is at the beginning of period. Enter 0 if payment is at the end of the period. Default is 0. If you enter a value other than 0 or 1, the Integration Service treats the value as 1.

Return Value

Numeric.

Example

The following expression returns 0.0077 as the monthly interest rate of a loan:

```
RATE( 48, -500, 20000 )
```

To calculate the annual interest rate of the loan, multiply 0.0077 by 12. The annual interest rate is 0.0924 or 9.24%.

REG_EXTRACT

Availability:
Designer
Workflow Manager

Extracts subpatterns of a regular expression within an input value. For example, from a regular expression pattern for a full name, you can extract the first name or last name.

Note: Use the REG_REPLACE function to replace a character pattern in a string with another character pattern.

Syntax

```
REG_EXTRACT( subject, 'pattern', subPatternNum )
```

Argument	Required/ Optional	Description
subject	Required	String datatype. Passes the value you want to compare against the regular expression pattern.
pattern	Required	String datatype. Regular expression pattern that you want to match. You must use perl compatible regular expression syntax. Enclose the pattern in single quotation marks. Enclose each subpattern in parentheses.
subPatternNum	Optional	Integer value. Subpattern number of the regular expression you want to match. Use the following guidelines to determine the subpattern number: - no value or 1. Extracts the first regular expression subpattern. - 2. Extracts the second regular expression subpattern. - n. Extracts the *n*th regular expression subpattern. Default is 1.

Using perl Compatible Regular Expression Syntax

You must use perl compatible regular expression syntax with REG_EXTRACT, REG_MATCH and REG_REPLACE functions.

The following table provides perl compatible regular expression syntax guidelines:

Syntax	Description
. (a period)	Matches any one character.
[a-z]	Matches one instance of a character in lower case. For example, [a-z] matches ab. Use [A-Z] to match characters in upper case.
\d	Matches one instance of any digit from 0-9.
\s	Matches a whitespace character.
\w	Matches one alphanumeric character, including underscore (_)
()	Groups an expression. For example, the parentheses in (\d-\d-\d\d) groups the expression \d\d-\d\d, which finds any two numbers followed by a hyphen and any two numbers, as in 12-34.
{}	Matches the number of characters. For example, \d{3} matches any three numbers, such as 650 or 510. Or, [a-z]{2} matches any two letters, such as CA or NY.
?	Matches the preceding character or group of characters zero or one time. For example, \d{3}(-{d{4})? matches any three numbers, which can be followed by a hyphen and any four numbers.
* (an asterisk)	Matches zero or more instances of the values that follow the asterisk. For example, *0 is any value that precedes a 0.
+	Matches one or more instances of the values that follow the plus sign. For example, \w+ is any value that follows an alphanumeric character.

For example, the following regular expression finds 5-digit U.S.A. zip codes, such as 93930, and 9-digit zip codes, such as 93930-5407:

```
\d{5}(-\d{4})?
```

\d{5} refers to any five numbers, such as 93930. The parentheses surrounding -\d{4} group this segment of the expression. The hyphen represents the hyphen of a 9-digit zip code, as in 93930-5407. \d{4} refers to any four numbers, such as 5407. The question mark states that the hyphen and last four digits are optional or can appear one time.

Converting COBOL Syntax to perl Compatible Regular Expression Syntax

If you are familiar with COBOL syntax, you can use the following information to write perl compatible regular expressions.

The following table shows examples of COBOL syntax and their perl equivalents:

COBOL Syntax	perl Syntax	Description
9	\d	Matches one instance of any digit from 0-9.
9999	\d\d\d\d or \d{4}	Matches any four digits from 0-9, as in 1234 or 5936.
x	[a-z]	Matches one instance of a letter.
9xx9	\d[a-z][a-z]\d	Matches any number followed by two letters and another number, as in 1ab2.

Converting SQL Syntax to perl Compatible Regular Expression Syntax

If you are familiar with SQL syntax, you can use the following information to write perl compatible regular expressions.

The following table shows examples of SQL syntax and their perl equivalents:

SQL Syntax	perl Syntax	Description
%	. *	Matches any string.
A%	A.*	Matches the letter “A” followed by any string, as in Area.
_	. (a period)	Matches any one character.
A_	A.	Matches “A” followed by any one character, such as AZ.

Return Value

Returns the value of the *n*th subpattern that is part of the input value. The *nth* subpattern is based on the value you specify for subPatternNum.

NULL if the input is a null value or if the pattern is null.

Example

You might use REG_EXTRACT in an expression to extract middle names from a regular expression that matches first name, middle name, and last name. For example, the following expression returns the middle name of a regular expression:

```
REG_EXTRACT( Employee_Name, '(\w+)\s+(\w+)\s+(\w+)',2)
```

Employee_Name	Return Value
Stephen Graham Smith	Graham
Juan Carlos Fernando	Carlos

REG_MATCH

Availability:
Designer
Workflow Manager

Returns whether a value matches a regular expression pattern. This lets you validate data patterns, such as IDs, telephone numbers, postal codes, and state names.

Note: Use the REG_REPLACE function to replace a character pattern in a string with a new character pattern.

Syntax

```
REG_MATCH( subject, pattern )
```

Argument	Required/ Optional	Description
subject	Required	String datatype. Passes the value you want to match against the regular expression pattern.
pattern	Required	String datatype. Regular expression pattern that you want to match. You must use perl compatible regular expression syntax. Enclose the pattern in single quotes. For more information, see “REG_EXTRACT” on page 119.

Return Value

TRUE if the data matches the pattern.

FALSE if the data does not match the pattern.

NULL if the input is a null value or if the pattern is NULL.

Example

You might use REG_MATCH in an expression to validate telephone numbers. For example, the following expression matches a 10-digit telephone number against the pattern and returns a Boolean value based on the match:

```
REG_MATCH (Phone_Number, '(\d\d\d-\d\d\d-\d\d\d\d)' )
```

Phone_Number	Return Value
408-555-1212	TRUE
	NULL

Phone_Number	Return Value
510-555-1212	TRUE
92 555 51212	FALSE
650-555-1212	TRUE
415-555-1212	TRUE
831 555 12123	FALSE

Tip

You can also use REG_MATCH for the following tasks:

- To verify that a value matches a pattern. This use is similar to the SQL LIKE function.
- To verify that values are characters. This use is similar to the SQL IS_CHAR function.

To verify that a value matches a pattern, use a period (.) and an asterisk (*) with the REG_MATCH function in an expression. A period matches any one character. An asterisk matches 0 or more instances of values that follow it.

For example, use the following expression to find account numbers that begin with 1835:

```
REG_MATCH(ACCOUNT_NUMBER, '1835.*')
```

To verify that values are characters, use a REG_MATCH function with the regular expression [a-zA-Z]+. a-z matches all lowercase characters. A-Z matches all uppercase characters. The plus sign (+) indicates that there should be at least one character.

For example, use the following expression to verify that a list of last names contain only characters:

```
REG_MATCH(LAST_NAME, '[a-zA-Z]+')
```

REG_REPLACE

Availability:
Designer
Workflow Manager

Replaces characters in a string with a another character pattern. By default, REG_REPLACE searches the input string for the character pattern you specify and replaces all occurrences with the replacement pattern. You can also indicate the number of occurrences of the pattern you want to replace in the string.

Syntax

```
REG_REPLACE( subject, pattern, replace, numReplacements )
```

Argument	Required/ Optional	Description
subject	Required	String datatype. Passes the string you want to search.
pattern	Required	String datatype. Passes the character string to be replaced. You must use perl compatible regular expression syntax. Enclose the pattern in single quotes. For more information, see “REG_EXTRACT” on page 119.

Argument	Required/ Optional	Description
replace	Required	String datatype. Passes the new character string.
numReplacements	Optional	Numeric datatype. Specifies the number of occurrences you want to replace. If you omit this option, REG_REPLACE will replace all occurrences of the character string.

Return Value

String

Example

The following expression removes additional spaces from the Employee name data for each row of the Employee_name port:

```
REG_REPLACE( Employee_Name, '\s+', ' ')
```

Employee_Name	RETURN VALUE
Adam Smith	Adam Smith
Greg Sanders	Greg Sanders
Sarah Fe	Sarah Fe
Sam Cooper	Sam Cooper

REPLACECHR

Availability:
Designer
Workflow Manager

Replaces characters in a string with a single character or no character. REPLACECHR searches the input string for the characters you specify and replaces all occurrences of all characters with the new character you specify.

Syntax

```
REPLACECHR( CaseFlag, InputString, OldCharSet, NewChar )
```

Argument	Required/ Optional	Description
CaseFlag	Required	Must be an integer. Determines whether the arguments in this function are case sensitive. You can enter any valid transformation expression. When *CaseFlag* is a number other than 0, the function is case sensitive. When *CaseFlag* is a null value or 0, the function is not case sensitive.
InputString	Required	Must be a character string. Passes the string you want to search. You can enter any valid transformation expression. If you pass a numeric value, the function converts it to a character string. If *InputString* is NULL, REPLACECHR returns NULL.

Argument	Required/ Optional	Description
OldCharSet	Required	Must be a character string. The characters you want to replace. You can enter one or more characters. You can enter any valid transformation expression. You can also enter a text literal enclosed within single quotation marks, for example, 'abc'. If you pass a numeric value, the function converts it to a character string. If *OldCharSet* is NULL or empty, REPLACECHR returns *InputString*.
NewChar	Required	Must be a character string. You can enter one character, an empty string, or NULL. You can enter any valid transformation expression. If *NewChar* is NULL or empty, REPLACECHR removes all occurrences of all characters in *OldCharSet* in *InputString*. If *NewChar* contains more than one character, REPLACECHR uses the first character to replace *OldCharSet*.

Return Value

String.

Empty string if REPLACECHR removes all characters in *InputString*.

NULL if *InputString* is NULL.

InputString if *OldCharSet* is NULL or empty.

Examples

The following expression removes the double quotes from web log data for each row in the WEBLOG port:

```
REPLACECHR( 0, WEBLOG, '"', NULL )
```

WEBLOG	RETURN VALUE
"GET /news/index.html HTTP/1.1"	GET /news/index.html HTTP/1.1
"GET /companyinfo/index.html HTTP/1.1"	GET /companyinfo/index.html HTTP/1.1
GET /companyinfo/index.html HTTP/1.1	GET /companyinfo/index.html HTTP/1.1
NULL	NULL

The following expression removes multiple characters for each row in the WEBLOG port:

```
REPLACECHR ( 1, WEBLOG, '][', NULL )
```

WEBLOG	RETURN VALUE
[29/Oct/2001:14:13:50 -0700]	29/Oct/2001:14:13:50 -0700
[31/Oct/2000:19:45:46 -0700] "GET /news/index.html HTTP/1.1"	31/Oct/2000:19:45:46 -0700 GET /news/index.html HTTP/1.1
[01/Nov/2000:10:51:31 -0700] "GET /news/index.html HTTP/1.1"	01/Nov/2000:10:51:31 -0700 GET /news/index.html HTTP/1.1
NULL	NULL

The following expression changes part of the value of the customer code for each row in the CUSTOMER_CODE port:

```
REPLACECHR ( 1, CUSTOMER_CODE, 'A', 'M' )
```

CUSTOMER_CODE	RETURN VALUE
ABA	MBM
abA	abM
BBC	BBC
ACC	MCC
NULL	NULL

The following expression changes part of the value of the customer code for each row in the CUSTOMER_CODE port:

```
REPLACECHR ( 0, CUSTOMER_CODE, 'A', 'M' )
```

CUSTOMER_CODE	RETURN VALUE
ABA	MBM
abA	MbM
BBC	BBC
ACC	MCC

The following expression changes part of the value of the customer code for each row in the CUSTOMER_CODE port:

```
REPLACECHR ( 1, CUSTOMER_CODE, 'A', NULL )
```

CUSTOMER_CODE	RETURN VALUE
ABA	B
BBC	BBC
ACC	CC
AAA	*[empty string]*
aaa	aaa
NULL	NULL

The following expression removes multiple numbers for each row in the INPUT port:

```
REPLACECHR ( 1, INPUT, '14', NULL )
```

INPUT	RETURN VALUE
12345	235
4141	NULL
111115	5
NULL	NULL

When you want to use a single quote (') in either *OldCharSet* or *NewChar*, you must use the CHR function. The single quote is the only character that cannot be used inside a string literal.

The following expression removes multiple characters, including the single quote, for each row in the INPUT port:

```
REPLACECHR (1, INPUT, CHR(39), NULL )
```

INPUT	RETURN VALUE
'Tom Smith' 'Laura Jones'	Tom Smith Laura Jones
Tom's	Toms
NULL	NULL

REPLACESTR

Availability:
Designer
Workflow Manager

Replaces characters in a string with a single character, multiple characters, or no character. REPLACESTR searches the input string for all strings you specify and replaces them with the new string you specify.

Syntax

```
REPLACESTR ( CaseFlag, InputString, OldString1, [OldString2, ... OldStringN,] NewString )
```

Argument	Required/ Optional	Description
CaseFlag	Required	Must be an integer. Determines whether the arguments in this function are case sensitive. You can enter any valid transformation expression. When *CaseFlag* is a number other than 0, the function is case sensitive. When *CaseFlag* is a null value or 0, the function is not case sensitive.
InputString	Required	Must be a character string. Passes the strings you want to search. You can enter any valid transformation expression. If you pass a numeric value, the function converts it to a character string. If *InputString* is NULL, REPLACESTR returns NULL.
OldString	Required	Must be a character string. The string you want to replace. You must enter at least one *OldString* argument. You can enter one or more characters per *OldString* argument. You can enter any valid transformation expression. You can also enter a text literal enclosed within single quotation marks, for example, 'abc'. If you pass a numeric value, the function converts it to a character string. When REPLACESTR contains multiple *OldString* arguments, and one or more *OldString* arguments is NULL or empty, REPLACESTR ignores the *OldString* argument. When all *OldString* arguments are NULL or empty, REPLACESTR returns *InputString*. The function replaces the characters in the *OldString* arguments in the order they appear in the function. For example, if you enter multiple *OldString* arguments, the first *OldString* argument has precedence over the second *OldString* argument, and the second *OldString* argument has precedence over the third *OldString* argument. When REPLACESTR replaces a string, it places the cursor after the replaced characters in *InputString* before searching for the next match.
NewString	Required	Must be a character string. You can enter one character, multiple characters, an empty string, or NULL. You can enter any valid transformation expression. If *NewString* is NULL or empty, REPLACESTR removes all occurrences of *OldString* in *InputString*.

Return Value

String.

Empty string if REPLACESTR removes all characters in *InputString*.

NULL if *InputString* is NULL.

InputString if all *OldString* arguments are NULL or empty.

Examples

The following expression removes the double quotes and two different text strings from web log data for each row in the WEBLOG port:

```
REPLACESTR( 1, WEBLOG, '"', 'GET ', ' HTTP/1.1', NULL )
```

WEBLOG	RETURN VALUE
"GET /news/index.html HTTP/1.1"	/news/index.html
"GET /companyinfo/index.html HTTP/1.1"	/companyinfo/index.html
GET /companyinfo/index.html	/companyinfo/index.html
GET	*[empty string]*
NULL	NULL

The following expression changes the title for certain values for each row in the TITLE port:

```
REPLACESTR ( 1, TITLE, 'rs.', 'iss', 's.' )
```

TITLE	RETURN VALUE
Mrs.	Ms.
Miss	Ms.
Mr.	Mr.
MRS.	MRS.

The following expression changes the title for certain values for each row in the TITLE port:

```
REPLACESTR ( 0, TITLE, 'rs.', 'iss', 's.' )
```

TITLE	RETURN VALUE
Mrs.	Ms.
MRS.	Ms.

The following expression shows how the REPLACESTR function replaces multiple `OldString` arguments for each row in the INPUT port:

```
REPLACESTR ( 1, INPUT, 'ab', 'bc', '*' )
```

INPUT	RETURN VALUE
abc	*c
abbc	**
abbbbc	*bb*
bc	*

The following expression shows how the REPLACESTR function replaces multiple `OldString` arguments for each row in the INPUT port:

```
REPLACESTR ( 1, INPUT, 'ab', 'bc', 'b' )
```

INPUT	RETURN VALUE
ab	b
bc	b
abc	bc
abbc	bb
abbcc	bbc

When you want to use a single quote (') in either *OldString* or *NewString*, you must use the CHR function. Use both the CHR and CONCAT functions to concatenate a single quote onto a string. The single quote is the only character that cannot be used inside a string literal. Consider the following example:

```
CONCAT( 'Joan', CONCAT( CHR(39), 's car' ))
```

The return value is:

```
Joan's car
```

The following expression changes a string that includes the single quote, for each row in the INPUT port:

```
REPLACESTR ( 1, INPUT, CONCAT('it', CONCAT(CHR(39), 's' )), 'its' )
```

INPUT	RETURN VALUE
it's	its
mit's	mits
mits	mits
mits'	mits'

REVERSE

Availability:
Designer
Workflow Manager

Reverses the input string.

Syntax

```
REVERSE( string )
```

Argument	Required/ Optional	Description
string	Required	Any character value. Value you want to reverse.

Return Value

String. Reverse of the input value.

Example

The following expression reverses the numbers of the customer code:

```
REVERSE( CUSTOMER_CODE )
```

CUSTOMER_CODE	RETURN VALUE
0001	1000
0002	2000
0003	3000
0004	4000

ROUND (Dates)

Availability:
Designer
Workflow Manager

Rounds one part of a date. You can also use ROUND to round numbers.

This function can round the following parts of a date:

- **Year.** Rounds the year portion of a date based on the month.
- **Month.** Rounds the month portion of a date based on the day of the month.
- **Day.** Rounds the day portion of the date based on the time.
- **Hour.** Rounds the hour portion of the date based on the minutes in the hour.
- **Minute.** Rounds the minute portion of the date based on the seconds.
- **Second.** Rounds the second portion of the date based on the milliseconds.
- **Millisecond.** Rounds the millisecond portion of the date based on the microseconds.
- **Microsecond.** Rounds the microsecond portion of the date based on the nanoseconds.

The following table shows the conditions used by the ROUND expression and the return values:

Condition	Expression	Return Value
Month between January and June, function returns January 1 of the same year and sets time to 00:00:00.000000000.	`ROUND(4/16/1998 8:24:19, 'YY')`	01/01/1998 00:00:00.000000000
Month between July and December, function returns January 1 of next year and sets time to 00:00:00.000000000.	`ROUND(07/30/1998 2:30:55, 'YY')`	01/01/1999 00:00:00.000000000
Day of month between 1 and 15, function returns date to the first day of the input month and sets time to 00:00:00.000000000.	`ROUND(4/15/1998 8:24:19, 'MM')`	4/1/1998 00:00:00.000000000
Day of month between 16 and the last day of the month, function returns the first day of the next month and sets time to 00:00:00.000000000.	`ROUND(05/22/1998 10:15:29, 'MM')`	5/1/1998 00:00:00.000000000
Time between 00:00:00 (12 a.m.) and 11:59:59 a.m., function returns the current date and sets time to 00:00:00.000000000 (12 a.m.).	`ROUND(06/13/1998 2:30:45, 'DD')`	06/13/1998 00:00:00.000000000
Time 12:00:00 (12 p.m.) or later, function rounds the date to the next day and sets time to 00:00:00.000000000 (12 a.m.).	`ROUND(06/13/1998 22:30:45, 'DD')`	06/14/1998 00:00:00.000000000
Minute portion of time between 0 and 29 minutes, function returns the current hour, and sets the minutes, seconds, and subseconds to 0.	`ROUND(04/01/1998 11:29:35, 'HH')`	04/01/1998 11:00:00.000000000
Minute portion of the time 30 or greater, function returns the next hour, and sets the minutes, seconds, and subseconds to 0.	`ROUND(04/01/1998 13:39:00, 'HH')`	04/01/1998 14:00:00.000000000
Time between 0 and 29 seconds, function returns the current minutes, and sets the seconds and subseconds to 0.	`ROUND(05/22/1998 10:15:29, 'MI')`	05/22/1998 10:15:00.000000000
Time between 30 and 59 seconds, function returns the next minute and sets the seconds and subseconds to 0.	`ROUND(05/22/1998 10:15:30, 'MI')`	05/22/1998 10:16:00.000000000
Time between 0 to 499 milliseconds, function returns the current second, and sets the milliseconds to 0.	`ROUND(05/22/1998 10:15:29.499, 'SS')`	05/22/1998 10:15:29.000000000
Time between 500 to 999 milliseconds, function returns the next second, and sets the milliseconds to 0.	`ROUND(05/22/1998 10:15:29.500, 'SS')`	05/22/1998 10:15:30.000000000
Time between 0 and 499 microseconds, function returns the current millisecond, and sets the microseconds to 0.	`ROUND(05/22/1998 10:15:29.498125, 'MS')`	05/22/1998 10:15:29.498000000
Time between 500 to 999 microseconds, function returns the next millisecond, and sets the microseconds to 0.	`ROUND(05/22/1998 10:15:29.498785, 'MS')`	05/22/1998 10:15:29.499000000

Condition	Expression	Return Value
Time between 0 and 499 nanoseconds, function returns the current microsecond, and sets the nanoseconds to 0.	`ROUND(05/22/1998 10:15:29.498125345, 'US')`	05/22/1998 10:15:29.498125000
Time between 500 and 999 nanoseconds, function returns the next microsecond, and sets the nanoseconds to 0.	`ROUND(05/22/1998 10:15:29.498125876, 'US')`	05/22/1998 10:15:29.498126000

Syntax

```
ROUND( date [,format] )
```

Argument	Required/ Optional	Description
date	Required	Date/Time datatype. You can nest TO_DATE to convert strings to dates before rounding.
format	Optional	Enter a valid format string. This is the portion of the date that you want to round. You can round only one portion of the date. If you omit the format string, the function rounds the date to the nearest day.

Return Value

Date with the specified part rounded. ROUND returns a date in the same format as the source date. You can link the results of this function to any port with a Date/Time datatype.

NULL if you pass a null value to the function.

Examples

The following expressions round the year portion of dates in the DATE_SHIPPED port:

```
ROUND( DATE_SHIPPED, 'Y' )
ROUND( DATE_SHIPPED, 'YY' )
ROUND( DATE_SHIPPED, 'YYY' )
ROUND( DATE_SHIPPED, 'YYYY' )
```

DATE_SHIPPED	RETURN VALUE
Jan 15 1998 2:10:30AM	Jan 1 1998 12:00:00.000000000AM
Apr 19 1998 1:31:20PM	Jan 1 1998 12:00:00.000000000AM
Dec 20 1998 3:29:55PM	Jan 1 1999 12:00:00.000000000AM
NULL	NULL

The following expressions round the month portion of each date in the DATE_SHIPPED port:

```
ROUND( DATE_SHIPPED, 'MM' )
ROUND( DATE_SHIPPED, 'MON' )
ROUND( DATE_SHIPPED, 'MONTH' )
```

DATE_SHIPPED	RETURN VALUE
Jan 15 1998 2:10:30AM	Jan 1 1998 12:00:00.000000000AM
Apr 19 1998 1:31:20PM	May 1 1998 12:00:00.000000000AM
Dec 20 1998 3:29:55PM	Jan 1 1999 12:00:00.000000000AM
NULL	NULL

The following expressions round the day portion of each date in the DATE_SHIPPED port:

```
ROUND( DATE_SHIPPED, 'D' )
ROUND( DATE_SHIPPED, 'DD' )
ROUND( DATE_SHIPPED, 'DDD' )
```

```
ROUND( DATE_SHIPPED, 'DY' )
ROUND( DATE_SHIPPED, 'DAY' )
```

DATE_SHIPPED	RETURN VALUE
Jan 15 1998 2:10:30AM	Jan 15 1998 12:00:00.000000000AM
Apr 19 1998 1:31:20PM	Apr 20 1998 12:00:00.000000000AM
Dec 20 1998 3:29:55PM	Dec 21 1998 12:00:00.000000000AM
Dec 31 1998 11:59:59PM	Jan 1 1999 12:00:00.000000000AM
NULL	NULL

The following expressions round the hour portion of each date in the DATE_SHIPPED port:

```
ROUND( DATE_SHIPPED, 'HH' )
ROUND( DATE_SHIPPED, 'HH12' )
ROUND( DATE_SHIPPED, 'HH24' )
```

DATE_SHIPPED	RETURN VALUE
Jan 15 1998 2:10:31AM	Jan 15 1998 2:00:00.000000000AM
Apr 19 1998 1:31:20PM	Apr 19 1998 2:00:00.000000000PM
Dec 20 1998 3:29:55PM	Dec 20 1998 3:00:00.000000000PM
Dec 31 1998 11:59:59PM	Jan 1 1999 12:00:00.000000000AM
NULL	NULL

The following expression rounds the minute portion of each date in the DATE_SHIPPED port:

```
ROUND( DATE_SHIPPED, 'MI' )
```

DATE_SHIPPED	RETURN VALUE
Jan 15 1998 2:10:30AM	Jan 15 1998 2:11:00.000000000AM
Apr 19 1998 1:31:20PM	Apr 19 1998 1:31:00.000000000PM
Dec 20 1998 3:29:55PM	Dec 20 1998 3:30:00.000000000PM
Dec 31 1998 11:59:59PM	Jan 1 1999 12:00:00.000000000AM
NULL	NULL

ROUND (Numbers)

Availability:
Designer
Workflow Manager

Rounds numbers to a specified number of digits or decimal places. You can also use ROUND to round dates.

Syntax

```
ROUND( numeric_value [, precision] )
```

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. You can enter any valid transformation expression. Use operators to perform arithmetic before you round the values.
precision	Optional	Positive or negative integer. If you enter a positive *precision*, the function rounds to this number of decimal places. For example, ROUND(12.99, 1) returns 13.0 and ROUND(15.44, 1) returns 15.4. If you enter a negative *precision*, the function rounds this number of digits to the left of the decimal point, returning an integer. For example, ROUND(12.99, -1) returns 10 and ROUND(15.99, -1) returns 20. If you enter decimal *precision*, the function rounds to the nearest integer before evaluating the expression. For example, ROUND(12.99, 0.8) returns 13.0 because the function rounds 0.8 to 1 and then evaluates the expression. If you omit the *precision* argument, the function rounds to the nearest integer, truncating the decimal portion of the number. For example, ROUND(12.99) returns 13.

Return Value

Numeric value.

If one of the arguments is NULL, ROUND returns NULL.

Note: If the return value is a decimal with precision greater than 15, you can enable high precision in the session properties to ensure decimal precision up to 28 digits.

Examples

The following expression returns the values in the Price port rounded to three decimal places.

```
ROUND( PRICE, 3 )
```

PRICE	RETURN VALUE
12.9936	12.994
15.9949	15.995
-18.8678	-18.868
56.9561	56.956
NULL	NULL

You can round digits to the left of the decimal point by passing a negative integer in the *precision* argument:

```
ROUND( PRICE, -2 )
```

PRICE	RETURN VALUE
13242.99	13200.0
1435.99	1400.0
-108.95	-100.0
NULL	NULL

If you pass a decimal value in the *precision* argument, the Integration Service rounds it to the nearest integer before evaluating the expression:

```
ROUND( PRICE, 0.8 )
```

PRICE	RETURN VALUE
12.99	13.0

PRICE	RETURN VALUE
56.34	56.3
NULL	NULL

If you omit the *precision* argument, the function rounds to the nearest integer:

```
ROUND( PRICE )
```

PRICE	RETURN VALUE
12.99	13.0
-15.99	-16.0
-18.99	-19.0
56.95	57.0
NULL	NULL

Tip

You can also use ROUND to explicitly set the precision of calculated values and achieve expected results. When the Integration Service runs in low precision mode, it truncates the result of calculations if the precision of the value exceeds 15 digits. For example, you might want to process the following expression in low precision mode:

```
7/3 * 3 = 7
```

In this case, the Integration Service evaluates the left hand side of the expression as 6.999999999999999 because it truncates the result of the first division operation. The Integration Service evaluates the entire expression as FALSE. This may not be the result you expect.

To achieve the expected result, use ROUND to round the truncated result of the left hand side of the expression to the expected result. The Integration Service evaluates the following expression as TRUE:

```
ROUND(7/3 * 3) = 7
```

RPAD

Availability:
Designer
Workflow Manager

Converts a string to a specified length by adding blanks or characters to the end of the string.

Syntax

```
RPAD( first_string, length [,second_string] )
```

Argument	Required/ Optional	Description
first_string	Required	Any string value. The strings you want to change. You can enter any valid transformation expression.
length	Required	Must be a positive integer literal. Specifies the length you want each string to be.
second_string	Optional	Any string value. Passes the string you want to append to the right-side of the *first_string* values. Enclose the characters you want to add to the end of the string within single quotation marks, for example, 'abc'. This argument is case sensitive. If you omit the second string, the function pads the end of the first string with blanks.

Return Value

String of the specified length.

NULL if a value passed to the function is NULL or if length is a negative number.

Examples

The following expression returns the item name with a length of 16 characters, appending the string '.' to the end of each item name:

```
RPAD( ITEM_NAME, 16, '.')
```

ITEM_NAME	RETURN VALUE
Flashlight	Flashlight......
Compass	Compass.........
Regulator System	Regulator System
Safety Knife	Safety Knife....

RPAD counts the length from left to right. So, if the first string is longer than the length, RPAD truncates the string from right to left. For example, RPAD(‘alphabetical’, 5, ‘x’) would return the string ‘alpha’. RPAD uses a partial part of the *second_string* when necessary.

The following expression returns the item name with a length of 16 characters, appending the string ‘*..*’ to the end of each item name:

```
RPAD( ITEM_NAME, 16, '*..*' )
```

ITEM_NAME	RETURN VALUE
Flashlight	Flashlight*..**.
Compass	Compass*..**..**
Regulator System	Regulator System
Safety Knife	Safety Knife*..*

RTRIM

Availability:
Designer
Workflow Manager

Removes blanks or characters from the end of a string.

If you do not specify a *trim_set* parameter in the expression:

- In UNICODE mode, RTRIM removes both single- and double-byte spaces from the end of a string.
- In ASCII mode, RTRIM removes only single-byte spaces.

If you use RTRIM to remove characters from a string, RTRIM compares the *trim_set* to each character in the *string* argument, character-by-character, starting with the right side of the string. If the character in the string matches any character in the *trim_set*, RTRIM removes it. RTRIM continues comparing and removing characters until it fails to find a matching character in the *trim_set*. It returns the string without the matching characters.

Syntax

```
RTRIM( string [, trim_set] )
```

Argument	Required/ Optional	Description
string	Required	Any string value. Passes the values you want to trim. You can enter any valid transformation expression. Use operators to perform comparisons or concatenate strings before removing blanks from the end of a string.
trim_set	Optional	Any string value. Passes the characters you want to remove from the end of the string. You can also enter a text literal. However, you must enclose the characters you want to remove from the end of the string within single quotation marks, for example, 'abc'. If you omit the second string, the function removes blanks from the end of the first string. RTRIM is case sensitive.

Return Value

String. The string values with the specified characters in the *trim_set* argument removed.

NULL if a value passed to the function is NULL.

Example

The following expression removes the characters ‘re’ from the strings in the LAST_NAME port:

```
RTRIM( LAST_NAME, 're')
```

LAST_NAME	RETURN VALUE
Nelson	Nelson
Page	Pag
Osborne	Osborn
NULL	NULL
Sawyer	Sawy
H. Bender	H. Bend
Steadman	Steadman

RTRIM removes ‘e’ from Page even though ‘r’ is the first character in the *trim_set*. This is because RTRIM searches, character-by-character, for the set of characters you specify in the *trim_set* argument. If the last character in the string matches the first character in the *trim_set*, RTRIM removes it. If, however, the last character in the string does not match, RTRIM compares the second character in the *trim_set*. If the second from last character in the string matches the second character in the *trim_set*, RTRIM removes it, and so on. When the character in the string fails to match the *trim_set*, RTRIM returns the string and evaluates the next row.

In the last example, the last character in Nelson does not match any character in the *trim_set* argument, so RTRIM returns the string 'Nelson' and evaluates the next row.

Tips for RTRIM

Use RTRIM and LTRIM with || or CONCAT to remove leading and trailing blanks after you concatenate two strings.

You can also remove multiple sets of characters by nesting RTRIM. For example, if you want to remove trailing blanks and the character ‘t’ from the end of each string in a column of names, you might create an expression similar to the following:

```
RTRIM( RTRIM( NAMES ), 't' )
```

SETCOUNTVARIABLE

Availability:
Designer

Counts the rows evaluated by the function and increments the current value of a mapping variable based on the count. Increases the current value by one for each row marked for insertion. Decreases the current value by one for each row marked for deletion. Keeps the current value the same for each row marked for update or reject. Returns the new current value.

At the end of a successful session, the Integration Service saves the last current value to the repository. When used with a session that contains multiple partitions, the Integration Service generates different current values for each partition. At the end of the session, it determines the total count for all partitions and saves the total to the repository. Unless overridden, it uses the saved value as the initial value of the variable for the next time you use this session.

Use the SETCOUNTVARIABLE function only once for each mapping variable in a pipeline. The Integration Service processes variable functions as it encounters them in the mapping. The order in which the Integration Service encounters variable functions in the mapping may not be the same for every session run. This may cause inconsistent results when you use the same variable function multiple times in a mapping.

Use SETCOUNTVARIABLE with mapping variables with a Count aggregation type. Use SETCOUNTVARIABLE in the following transformations:

- Expression
- Filter
- Router
- Update Strategy

The Integration Service does not save the final value of a mapping variable to the repository when any of the following are true:

- The session fails to complete.
- The session is configured for a test load.
- The session is a debug session.
- The session runs in debug mode and is configured to discard session output.

Syntax

```
SETCOUNTVARIABLE( $$Variable )
```

Argument	Required/ Optional	Description
$$*Variable*	Required	Name of the mapping variable you want to set. Use mapping variables with a count aggregation type.

Return Value

The current value of the variable.

Example

You have a mapping that updates a slowly changing dimension table containing distributor information. The following expression counts the number of current distributors with the mapping variable $$CurrentDistributors and returns the current value to the CUR_DIST port. It increases the count by one for each inserted row, decreases the count for each deleted row, and keeps the count the same for all updated or rejected rows. The initial value of $ $CurrentDistributors from the previous session run is 23.

```
SETCOUNTVARIABLE ($$CurrentDistributors)
```

(row marked for...)	DIST_ID	DISTRIBUTOR	CUR_DIST
(update)	000015	MSD Inc.	23
(insert)	000024	Darkroom Co.	24
(insert)	000025	Howard's Supply	25
(update)	000003	JNR Ltd.	25
(delete)	000024	Darkroom Co.	24
(insert)	000026	Supply.com	25

At the end of the session, the Integration Service saves ‘25’ to the repository as the current value for $ $CurrentDistributors. The next time the session runs, the Integration Service evaluates the initial value to $ $CurrentDistributors to ‘25’.

The Integration Service saves the same value for $$CurrentDistributors to the repository for sessions with multiple partitions as for sessions with a single partition.

SET_DATE_PART

Availability:
Designer
Workflow Manager

Sets one part of a Date/Time value to a value you specify. With SET_DATE_PART, you can change the following parts of a date:

- **Year.** Change the year by entering a positive integer in the *value* argument. Use any of the year format strings: Y, YY, YYY, or YYYY to set the year. For example, the following expression changes the year to 2001 for all dates in the SHIP_DATE port:

```
SET_DATE_PART( SHIP_DATE, 'YY', 2001 )
```

- **Month.** Change the month by entering a positive integer between 1 and 12 (January=1 and December=12) in the *value* argument. Use any of the month format strings: MM, MON, MONTH to set the month. For example, the following expression changes the month to October for all dates in the SHIP_DATE port:

```
SET_DATE_PART( SHIP_DATE, 'MONTH', 10 )
```

- **Day.** Change the day by entering a positive integer between 1 and 31 (except for the months that have less than 31 days: February, April, June, September, and November) in the *value* argument. Use any of the month format strings (D, DD, DDD, DY, and DAY) to set the day. For example, the following expression changes the day to 10 for all dates in the SHIP_DATE port:

```
SET_DATE_PART( SHIP_DATE, 'DD', 10 )
```

- **Hour.** Change the hour by entering a positive integer between 0 and 24 (where 0=12AM, 12=12PM, and 24 =12AM) in the *value* argument. Use any of the hour format strings (HH, HH12, HH24) to set the hour. For example, the following expression changes the hour to 14:00:00 (or 2:00:00PM) for all dates in the SHIP_DATE port:

```
SET_DATE_PART( SHIP_DATE, 'HH', 14 )
```

- **Minute.** Change the minutes by entering a positive integer between 0 and 59 in the *value* argument. Use the MI format string to set the minute. For example, the following expression changes the minute to 25 for all dates in the SHIP_DATE port:

```
SET_DATE_PART( SHIP_DATE, 'MI', 25 )
```

- **Seconds.** Change the seconds by entering a positive integer between 0 and 59 in the *value* argument. Use the SS format string to set the second. For example, the following expression changes the second to 59 for all dates in the SHIP_DATE port:

```
SET_DATE_PART( SHIP_DATE, 'SS', 59 )
```

- **Milliseconds**. Change the milliseconds by entering a positive integer between 0 and 999 in the *value* argument. Use the MS format string to set the milliseconds. For example, the following expression changes the milliseconds to 125 for all dates in the SHIP_DATE port:

```
SET_DATE_PART( SHIP_DATE, 'MS', 125 )
```

- **Microseconds.** Change the microseconds by entering a positive integer between 1000 and 999999 in the *value* argument. Use the US format string to set the microseconds. For example, the following expression changes the microseconds to 12555 for all dates in the SHIP_DATE port:

```
SET_DATE_PART( SHIP_DATE, 'US', 12555 )
```

- **Nanoseconds.** Change the nanoseconds by entering a positive integer between 1000000 and 999999999 in the *value* argument. Use the NS format string to set the nanoseconds. For example, the following expression changes the nanoseconds to `12555555` for all dates in the SHIP_DATE port:

```
SET_DATE_PART( SHIP_DATE, 'NS', 12555555 )
```

Syntax

```
SET_DATE_PART( date, format, value )
```

Argument	Required/ Optional	Description
date	Required	Date/Time datatype. The date you want to modify. You can enter any valid transformation expression.
format	Required	Format string specifying the portion of the date to be changed. The format string is not case sensitive.
value	Required	A positive integer value assigned to the specified portion of the date. The integer must be a valid value for the part of the date you want to change. If you enter an improper value such as February 30, the session fails.

Return Value

Date in the same format as the source date with the specified part changed.

NULL if a value passed to the function is NULL.

Examples

The following expressions change the hour to 4PM for each date in the DATE_PROMISED port:

```
SET_DATE_PART( DATE_PROMISED, 'HH', 16 )
SET_DATE_PART( DATE_PROMISED, 'HH12', 16 )
SET_DATE_PART( DATE_PROMISED, 'HH24', 16 )
```

DATE_PROMISED	RETURN VALUE
Jan 1 1997 12:15:56AM	Jan 1 1997 4:15:56PM
Feb 13 1997 2:30:01AM	Feb 13 1997 4:30:01PM
Mar 31 1997 5:10:15PM	Mar 31 1997 4:10:15PM
Dec 12 1997 8:07:33AM	Dec 12 1997 4:07:33PM
NULL	NULL

The following expressions change the month to June for the dates in the DATE_PROMISED port. The Integration Service displays an error when you try to create a date that does not exist, such as changing March 31 to June 31:

```
SET_DATE_PART( DATE_PROMISED, 'MM', 6 )
SET_DATE_PART( DATE_PROMISED, 'MON', 6 )
SET_DATE_PART( DATE_PROMISED, 'MONTH', 6 )
```

DATE_PROMISED	RETURN VALUE
Jan 1 1997 12:15:56AM	Jun 1 1997 12:15:56AM
Feb 13 1997 2:30:01AM	Jun 13 1997 2:30:01AM
Mar 31 1997 5:10:15PM	*Error. Integration Service doesn't write row.*
Dec 12 1997 8:07:33AM	Jun 12 1997 8:07:33AM
NULL	NULL

The following expressions change the year to 2000 for the dates in the DATE_PROMISED port:

```
SET_DATE_PART( DATE_PROMISED, 'Y', 2000 )
SET_DATE_PART( DATE_PROMISED, 'YY', 2000 )
SET_DATE_PART( DATE_PROMISED, 'YYY', 2000 )
SET_DATE_PART( DATE_PROMISED, 'YYYY', 2000 )
```

DATE_PROMISED	RETURN VALUE
Jan 1 1997 12:15:56AM	Jan 1 2000 12:15:56AM
Feb 13 1997 2:30:01AM	Feb 13 2000 2:30:01AM
Mar 31 1997 5:10:15PM	Mar 31 2000 5:10:15PM

DATE_PROMISED	RETURN VALUE
Dec 12 1997 8:07:33AM	Dec 12 2000 4:07:33PM
NULL	NULL

Tip

If you want to change multiple parts of a date at one time, you can nest multiple SET_DATE_PART functions within the *date* argument. For example, you might write the following expression to change all of the dates in the DATE_ENTERED port to July 1 1998:

```
SET_DATE_PART( SET_DATE_PART( SET_DATE_PART( DATE_ENTERED, 'YYYY', 1998),MM', 7), 'DD', 1)
```

SETMAXVARIABLE

Availability:
Designer

Sets the current value of a mapping variable to the higher of two values: the current value of the variable or the value you specify. Returns the new current value. The function executes only if a row is marked as insert. SETMAXVARIABLE ignores all other row types and the current value remains unchanged.

At the end of a successful session, the Integration Service saves the final current value to the repository. When used with a session that contains multiple partitions, the Integration Service generates different current values for each partition. At the end of the session, it saves the highest current value across all partitions to the repository. Unless overridden, it uses the saved value as the initial value of the variable for the next session run.

When used with a string mapping variable, SETMAXVARIABLE returns the higher string based on the sort order selected for the session.

Use the SETMAXVARIABLE function only once for each mapping variable in a pipeline. The Integration Service processes variable functions as it encounters them in the mapping. The order in which the Integration Service encounters variable functions in the mapping may not be the same for every session run. This can cause inconsistent results when you use the same variable function multiple times in a mapping.

Use SETMAXVARIABLE with mapping variables with a Max aggregation type. Use SETMAXVARIABLE in the following transformations:

- Expression
- Filter
- Router
- Update Strategy

The Integration Service does not save the final value of a mapping variable to the repository when any of the following conditions are true:

- The session fails to complete.
- The session is configured for a test load.
- The session is a debug session.
- The session runs in debug mode and is configured to discard session output.

Syntax

```
SETMAXVARIABLE( $$Variable, value )
```

Argument	Required/ Optional	Description
$$*Variable*	Required	Name of the mapping variable you want to set. Use mapping variables with Max aggregation type.
value	Required	The value you want the Integration Service to compare against the current value of the variable. You can enter any valid transformation expression that evaluates to a datatype compatible with the datatype of the variable.

Return Value

The higher of two values: the current value of the variable or the value you specified. The return value is the new current value of the variable.

When *value* is NULL the Integration Service returns the current value of *$$Variable*.

Examples

The following expression compares the number of items purchased in each transaction with a mapping variable $$MaxItems. It sets $$MaxItems to the higher of two values and returns the historically highest number of items purchased in a single transaction to the MAX_ITEMS port. The initial value of $$MaxItems from the previous session run is 22.

```
SETMAXVARIABLE ($$MAXITEMS, ITEMS)
```

TRANSACTION	ITEMS	MAX_ITEMS
0100002	12	22
0100003	5	22
0100004	18	22
0100005	35	35
0100006	5	35
0100007	14	35

At the end of the session, the Integration Service saves ‘35’ to the repository as the maximum current value for $$MaxItems. The next time the session runs, the Integration Service evaluates the initial value to $$MaxItems to ‘35’.

If the same session contains three partitions, the Integration Service evaluates $$MaxItems for each partition. Then, it saves the largest value to the repository. For example, the last evaluated value for $$MaxItems in each partition is as follows:

Partition	Final Current Value for $$MaxItems
Partition 1	35
Partition 2	23
Partition 3	22

SETMINVARIABLE

Availability:
Designer

Sets the current value of a mapping variable to the lower of two values: the current value of the variable or the value you specify. Returns the new current value. The SETMINVARIABLE function executes only if a row is marked as insert. SETMINVARIABLE ignores all other row types and the current value remains unchanged.

At the end of a successful session, the Integration Service saves the final current value to the repository. When used with a session that contains multiple partitions, the Integration Service generates different current values for each partition. At the end of the session, it saves the lowest current value across all partitions to the repository. Unless overridden, it uses the saved value as the initial value of the variable for the next session run.

When used with a string mapping variable, SETMINVARIABLE returns the lower string based on the sort order selected for the session.

Use the SETMINVARIABLE function only once for each mapping variable in a pipeline. The Integration Service processes variable functions as it encounters them in the mapping. The order in which the Integration Service encounters variable functions in the mapping may not be the same for every session run. This may cause inconsistent results when you use the same variable function multiple times in a mapping.

Use SETMINVARIABLE with mapping variables with a Min aggregation type. Use SETMINVARIABLE in the following transformations:

- Expression
- Filter
- Router
- Update Strategy

The Integration Service does not save the final value of a mapping variable to the repository when any of the following conditions are true:

- The session fails to complete.
- The session is configured for a test load.
- The session is a debug session.
- The session runs in debug mode and is configured to discard session output.

Syntax

```
SETMINVARIABLE( $$Variable, value )
```

Argument	Required/ Optional	Description
$$*Variable*	Required	Name of the mapping variable you want to set. Use with mapping variables with Min aggregation type.
value	Required	The value you want the Integration Service to compare against the current value of the variable. You can enter any valid transformation expression that evaluates to a datatype compatible with the datatype of the variable.

Return Value

The lower of two values: the current value of the variable or the value you specified. The return value is the new current value of the variable.

When *value* is NULL, the Integration Service returns the current value of *$$Variable*.

Example

The following expression compares the price of an item with a mapping variable $$MinPrice. It sets $$MinPrice to the lower of two values and returns the historically lowest item price to the MIN_PRICE port. The initial value of $$MinPrice from the previous session run is 22.50.

```
SETMINVARIABLE ($$MinPrice, PRICE)
```

DATE	PRICE	MIN_PRICE
05/01/2000 09:00:00	23.50	22.50
05/01/2000 10:00:00	27.00	22.50
05/01/2000 11:00:00	26.75	22.50
05/01/2000 12:00:00	25.25	22.50
05/01/2000 13:00:00	22.00	22.00
05/01/2000 14:00:00	22.75	22.00
05/01/2000 15:00:00	23.00	22.00
05/01/2000 16:00:00	24.25	22.00
05/01/2000 17:00:00	24.00	22.00

At the end of the session, the Integration Service saves 22.00 to the repository as the minimum current value for $$MinPrice. The next time the session runs, the Integration Service evaluates the initial value to $$MinPrice to 22.00.

If the same session contains three partitions, the Integration Service evaluates $$MinPrice for each partition. Then, it saves the smallest value to the repository. For example, the last evaluated value for $$MinPrice in each partition is as follows:

Partition	Final Current Value for $$MinPrice
Partition 1	22.00
Partition 2	22.50
Partition 3	22.50

SETVARIABLE

Availability:
Designer

Sets the current value of a mapping variable to a value you specify. Returns the specified value. The SETVARIABLE function executes only if a row is marked as insert or update. SETVARIABLE ignores all other row types and the current value remains unchanged.

At the end of a successful session, the Integration Service compares the final current value of the variable to the start value of the variable. Based on the aggregate type of the variable, it saves a final current value to the repository. Unless overridden, it uses the saved value as the initial value of the variable for the next session run.

Use the SETVARIABLE function only once for each mapping variable in a pipeline. The Integration Service processes variable functions as it encounters them in the mapping. The order in which the Integration Service encounters variable functions in the mapping may not be the same for every session run. This may cause inconsistent results when you use the same variable function multiple times in a mapping.

Use SETVARIABLE in the following transformations:

- Expression
- Filter

- Router
- Update Strategy

The Integration Service does not save the final value of a mapping variable to the repository when any of the following conditions are true:

- The session fails to complete.
- The session is configured for a test load.
- The session is a debug session.
- The session runs in debug mode and is configured to discard session output.

Syntax

```
SETVARIABLE( $$Variable, value )
```

Argument	Required/ Optional	Description
$$*Variable*	Required	Name of the mapping variable you want to set. Use with mapping variables with Max/Min aggregation type.
value	Required	The value you want to set the current value of the variable to. You can enter any valid transformation expression that evaluates to a datatype compatible with the datatype of the variable.

Return Value

Current value of the variable.

When *value* is NULL, the Integration Service returns the current value of *$$Variable*.

Examples

The following expression sets a mapping variable $$Time to the system date at the time the Integration Service evaluates the row and returns the system date to the SET_$$TIME port:

```
SETVARIABLE ($$Time, SYSDATE)
```

```
TRANSACTION                 TOTAL           SET_$$TIME
0100002                     534.23          10/10/2000 01:34:33
0100003                     699.01          10/10/2000 01:34:34
0100004                     97.50           10/10/2000 01:34:35
0100005                     116.43          10/10/2000 01:34:36
0100006                     323.95          10/10/2000 01:34:37
```

At the end of the session, the Integration Service saves 10/10/2000 01:34:37 to the repository as the last evaluated current value for $$Time. The next time the session runs, the Integration Service evaluates all references to $$Time to 10/10/2000 01:34:37.

The following expression sets the mapping variable $$Timestamp to the timestamp associated with the row and returns the timestamp to the SET_$$TIMESTAMP port:

```
SETVARIABLE ($$Time, TIMESTAMP)
```

```
TRANSACTION         TIMESTAMP                       TOTAL       SET_$$TIMESTAMP
0100002             10/01/2000 12:01:01             534.23      10/01/2000 12:01:01
0100003             10/01/2000 12:10:22             699.01      10/01/2000 12:10:22
0100004             10/01/2000 12:16:45             97.50       10/01/2000 12:16:45
0100005             10/01/2000 12:23:10             116.43      10/01/2000 12:23:10
0100006             10/01/2000 12:40:31             323.95      10/01/2000 12:40:31
```

At the end of the session, the Integration Service saves 10/01/2000 12:40:31 to the repository as the last evaluated current value for $$Timestamp.

The next time the session runs, the Integration Service evaluates the initial value of $$Timestamp to 10/01/2000 12:40:31.

SIGN

Availability:
Designer
Workflow Manager

Returns whether a numeric value is positive, negative, or 0.

Syntax

```
SIGN( numeric_value )
```

Argument	Required/ Optional	Description
numeric_value	Required	Numeric value. Passes the values you want to evaluate. You can enter any valid transformation expression.

Return Value

-1 for negative values.

0 for 0.

1 for positive values.

NULL if NULL.

Example

The following expression determines if the SALES port includes any negative values:

```
SIGN( SALES )
```

SALES	RETURN VALUE
100	1
-25.99	-1
0	0
NULL	NULL

SIN

Availability:
Designer
Workflow Manager

Returns the sine of a numeric value (expressed in radians).

Syntax

```
SIN( numeric_value )
```

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. Numeric data expressed in radians (degrees multiplied by pi divided by 180). Passes the values for which you want to calculate the sine. You can enter any valid transformation expression. You can also use operators to convert a numeric value to radians or perform arithmetic within the SIN calculation.

Return Value

Double value.

NULL if a value passed to the function is NULL.

Example

The following expression converts the values in the Degrees port to radians and then calculates the sine for each radian:

```
SIN( DEGREES * 3.14159265359 / 180 )
```

DEGREES	RETURN VALUE
0	0
90	1
70	0.939692620785936
30	0.50000000000003
5	0.0871557427476639
89	0.999847695156393
NULL	NULL

You can perform arithmetic on the values passed to SIN before the function calculates the sine. For example:

```
SIN( ARCS * 3.14159265359 / 180 )
```

SINH

Availability:
Designer
Workflow Manager

Returns the hyperbolic sine of the numeric value.

Syntax

```
SINH( numeric_value )
```

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. Numeric data expressed in radians (degrees multiplied by pi divided by 180). Passes the values for which you want to calculate the hyperbolic sine. You can enter any valid transformation expression.

Return Value

Double value.

NULL if a value passed to the function is NULL.

Example

The following expression returns the hyperbolic sine for the values in the Angles port:

```
SINH( ANGLES )
```

ANGLES	RETURN VALUE
1.0	1.1752011936438
2.897	9.03225804884884
3.66	19.4178051793031
5.45	116.376934801486
0	0.0
0.345	0.35188478309993
NULL	NULL

Tip

You can perform arithmetic on the values passed to SINH before the function calculates the hyperbolic sine. For example:

```
SINH( MEASURES.ARCS / 180 )
```

SOUNDEX

Availability:
Designer
Workflow Manager

Encodes a string value into a four-character string.

SOUNDEX works for characters in the English alphabet (A-Z). It uses the first character of the input string as the first character in the return value and encodes the remaining three unique consonants as numbers.

SOUNDEX encodes characters according to the following list of rules:

- Uses the first character in *string* as the first character in the return value and encodes it in uppercase. For example, both SOUNDEX('John') and SOUNDEX('john') return 'J500'.
- Encodes the first three unique consonants following the first character in *string* and ignores the rest. For example, both SOUNDEX('JohnRB') and SOUNDEX('JohnRBCD') return 'J561'.

- Assigns a single code to consonants that sound alike.

  The following table lists SOUNDEX encoding guidelines for consonants:

Table 2. SOUNDEX Encoding Guidelines for Consonants

Code	Consonant
1	B, P, F, V
2	C, S, G, J, K, Q, X, Z
3	D, T
4	L
5	M, N
6	R

- Skips the characters A, E, I, O, U, H, and W unless one of them is the first character in *string*. For example, SOUNDEX('A123') returns 'A000' and SOUNDEX('MAeiouhwC') returns 'M000'.
- If *string* produces fewer than four characters, SOUNDEX pads the resulting string with zeroes. For example, SOUNDEX('J') returns 'J000'.
- If *string* contains a set of consecutive consonants that use the same code listed in "SOUNDEX" on page 148, SOUNDEX encodes the first occurrence and skips the remaining occurrences in the set. For example, SOUNDEX('AbbpdMN') returns 'A135'.
- Skips numbers in *string*. For example, both SOUNDEX('Joh12n') and SOUNDEX('1John') return 'J500'.
- Returns NULL if *string* is NULL or if all the characters in *string* are not letters of the English alphabet.

Syntax

```
SOUNDEX( string )
```

Argument	Required/ Optional	Description
string	Required	Character string. Passes the string value you want to encode. You can enter any valid transformation expression.

Return Value

String.

NULL if one of the following conditions is true:

- If value passed to the function is NULL.
- No character in *string* is a letter of the English alphabet.
- *string* is empty.

Example

The following expression encodes the values in the EMPLOYEE_NAME port:

```
SOUNDEX( EMPLOYEE_NAME )
```

EMPLOYEE_NAME	RETURN VALUE
John	J500
William	W450
jane	J500
joh12n	J500
1abc	A120
NULL	NULL

SQRT

Availability:
Designer
Workflow Manager

Returns the square root of a non-negative numeric value.

Syntax

```
SQRT( numeric_value )
```

Argument	Required/ Optional	Description
numeric_value	Required	Positive numeric value. Passes the values for which you want to calculate a square root. You can enter any valid transformation expression.

Return Value

Double value.

NULL if a value passed to the function is NULL.

Example

The following expression returns the square root for the values in the Numbers port:

```
SQRT( NUMBERS )
```

NUMBERS	RETURN VALUE
100	10
-100	*Error. Integration Service does not write row.*
NULL	NULL
60.54	7.78074546557076

The value -100 results in an error during the session, since the function SQRT only evaluates positive numeric values. If you pass a negative value or character value, the Integration Service displays a Transformation Evaluation Error and does not write the row.

You can perform arithmetic on the values passed to SQRT before the function calculates the square root.

STDDEV

Availability:
Designer

Returns the standard deviation of the numeric values you pass to this function. STDDEV is used to analyze statistical data. You can nest only one other aggregate function within STDDEV, and the nested function must return a Numeric datatype.

Syntax

```
STDDEV( numeric_value [,filter_condition] )
```

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatypes. This function passes the values for which you want to calculate a standard deviation or the results of a function. You can enter any valid transformation expression. You can use operators to average values in different ports.
filter_condition	Optional	Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression.

Return Value

Numeric value.

NULL if all values passed to the function are NULL or if no rows are selected (for example, the filter condition evaluates to FALSE or NULL for all rows).

Note: If the return value is decimal with precision greater than 15, you can enable high precision in the session properties to ensure decimal precision up to 28 digits.

Nulls

If a single value is NULL, STDDEV ignores it. However, if all values are NULL, STDDEV returns NULL.

Note: By default, the Integration Service treats null values as NULLS in aggregate functions. If you pass an entire port or group of null values, the function returns NULL. However, when you configure the Integration Service, you can choose how you want to handle null values in aggregate functions. You can treat null values as 0 in aggregate functions or as NULL.

Group By

STDDEV groups values based on group by ports you define in the transformation, returning one result for each group.

If there is no group by port, STDDEV treats all rows as one group, returning one value.

Examples

The following expression calculates the standard deviation of all rows greater than $2000.00 in the TOTAL_SALES port:

```
STDDEV( SALES, SALES > 2000.00 )
```

```
SALES
2198.0
1010.90
2256.0
```

```
SALES
153.88
3001.0
NULL
8953.0
RETURN VALUE: 3254.60361129688
```

The function does not include the values 1010.90 and 153.88 in the calculation because the *filter_condition* specifies sales greater than $2,000.

The following expression calculates the standard deviation of all rows in the SALES port:

```
STDDEV(SALES)
```

```
SALES
2198.0
2198.0
2198.0
2198.0
RETURN VALUE: 0
```

The return value is 0 because each row contains the same number (no standard deviation exists). If there is no standard deviation, the return value is 0.

SUBSTR

Availability:
Designer
Workflow Manager

Returns a portion of a string. SUBSTR counts all characters, including blanks, starting at the beginning of the string.

Syntax

```
SUBSTR( string, start [,length] )
```

Argument	Required/ Optional	Description
string	Required	Must be a character string. Passes the strings you want to search. You can enter any valid transformation expression. If you pass a numeric value, the function converts it to a character string.
start	Required	Must be an integer. The position in the string where you want to start counting. You can enter any valid transformation expression. If the start position is a positive number, SUBSTR locates the start position by counting from the beginning of the string. If the start position is a negative number, SUBSTR locates the start position by counting from the end of the string. If the start position is 0, SUBSTR searches from the first character in the string.
length	Optional	Must be an integer greater than 0. The number of characters you want SUBSTR to return. You can enter any valid transformation expression. If you omit the length argument, SUBSTR returns all of the characters from the start position to the end of the string. If you pass a negative integer or 0, the function returns an empty string. If you pass a decimal, the function rounds it to the nearest integer value.

Return Value

String.

Empty string if you pass a negative or 0 length value.

NULL if a value passed to the function is NULL.

Examples

The following expressions return the area code for each row in the Phone port:

```
SUBSTR( PHONE, 0, 3 )
```

PHONE	RETURN VALUE
809-555-0269	809
357-687-6708	357
NULL	NULL

```
SUBSTR( PHONE, 1, 3 )
```

PHONE	RETURN VALUE
809-555-3915	809
357-687-6708	357
NULL	NULL

The following expressions return the phone number without the area code for each row in the Phone port:

```
SUBSTR( PHONE, 5, 8 )
```

PHONE	RETURN VALUE
808-555-0269	555-0269
809-555-3915	555-3915
357-687-6708	687-6708
NULL	NULL

You can also pass a negative start value to return the phone number for each row in the Phone port. The expression still reads the source string from left to right when returning the result of the *length* argument:

```
SUBSTR( PHONE, -8, 3 )
```

PHONE	RETURN VALUE
808-555-0269	555
809-555-3915	555
357-687-6708	687
NULL	NULL

You can nest INSTR in the *start* or *length* argument to search for a specific string and return its position.

The following expression evaluates a string, starting from the end of the string. The expression finds the last (right-most) space in the string and then returns all characters preceding it:

```
SUBSTR( CUST_NAME,1,INSTR( CUST_NAME,' ' ,-1,1 ) - 1 )
```

CUST_NAME	RETURN VALUE
PATRICIA JONES	PATRICIA
MARY ELLEN SHAH	MARY ELLEN

The following expression removes the character '#' from a string:

```
SUBSTR( CUST_ID, 1, INSTR(CUST_ID, '#')-1 ) || SUBSTR( CUST_ID, INSTR(CUST_ID, '#')+1 )
```

When the *length* argument is longer than the string, SUBSTR returns all the characters from the start position to the end of the string. Consider the following example:

```
SUBSTR('abcd', 2, 8)
```

The return value is ‘bcd’. Compare this result to the following example:

```
SUBSTR('abcd', -2, 8)
```

The return value is ‘cd’.

SUM

Availability:
Designer

Returns the sum of all values in the selected port. Optionally, you can apply a filter to limit the rows you read to calculate the total. You can nest only one other aggregate function within SUM, and the nested function must return a Numeric datatype.

Syntax

```
SUM( numeric_value [, filter_condition ] )
```

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. Passes the values you want to add. You can enter any valid transformation expression. You can use operators to add values in different ports.
filter_condition	Optional	Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression.

Return Value

Numeric value.

NULL if all values passed to the function are NULL or if no rows are selected (for example, the filter condition evaluates to FALSE or NULL for all rows).

Note: If the return value is decimal with precision greater than 15, you can enable high precision in the session properties to ensure decimal precision up to 28 digits.

Nulls

If a single value is NULL, SUM ignores it. However, if all values passed from the port are NULL, SUM returns NULL.

Note: By default, the Integration Service treats null values as NULLS in aggregate functions. If you pass an entire port or group of null values, the function returns NULL. However, when you configure the Integration Service, you can choose how you want to handle null values in aggregate functions. You can treat null values as 0 in aggregate functions or as NULL.

Group By

SUM groups values based on group by ports you define in the transformation, returning one result for each group.

If there is no group by port, SUM treats all rows as one group, returning one value.

Example

The following expression returns the sum of all values greater than 2000 in the Sales port:

```
SUM( SALES, SALES > 2000 )
```

```
SALES
2500.0
1900.0
1200.0
NULL
3458.0
4519.0
RETURN VALUE: 10477.0
```

Tip

You can perform arithmetic on the values passed to SUM before the function calculates the total. For example:

```
SUM( QTY * PRICE - DISCOUNT )
```

SYSTIMESTAMP

Availability:
Designer
Workflow Manager

Returns the current date and time of the node hosting the Integration Service with precision to the nanosecond. The precision to which you display the date and time depends on the platform.

The return value of the function varies depending on how you configure the argument:

- When you configure the argument of SYSTIMESTAMP as a variable, the Integration Service evaluates the function for each row in the transformation.
- When you configure the argument of SYSTIMESTAMP as a constant, the Integration Service evaluates the function once and retains the value for each row in the transformation.

Syntax

```
SYSTIMESTAMP( [format] )
```

Argument	Required/ Optional	Description
format	Optional	Precision to which you want to retrieve the timestamp. You can specify precision up to seconds (SS), milliseconds (MS), microseconds (US), or nanoseconds (NS). Enclose the format string within single quotation marks. The format string is not case sensitive. For example, to display the date and time to the precision of milliseconds use the following syntax: SYSTIMESTAMP('MS'). Default precision is microseconds (US).

Return Value

Timestamp. Returns date and time to the specified precision.

Examples

Your organization has an online order service and processes real-time data. You can use the SYSTIMESTAMP function to generate a primary key for each transaction in the target database.

Create an Expression transformation with the following ports and values:

Port Name	Port Type	Expression
Customer_Name	Input	n/a
Order_Qty	Input	n/a
Time_Counter	Variable	'US'
Transaction_Id	Output	SYSTIMESTAMP (Time_Counter)

At run time, the Integration Service generates the system time to the precision of microseconds for each row:

Customer_Name	Order_Qty	Transaction_Id
Vani Deed	14	07/06/2007 18:00:30.701015000
Kalia Crop	3	07/06/2007 18:00:30.701029000
Vani Deed	6	07/06/2007 18:00:30.701039000
Harry Spoon	32	07/06/2007 18:00:30.701048000

TAN

Availability:
Designer
Workflow Manager

Returns the tangent of a numeric value (expressed in radians).

Syntax

```
TAN( numeric_value )
```

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. Numeric data expressed in radians (degrees multiplied by pi divided by 180). Passes the numeric values for which you want to calculate the tangent. You can enter any valid transformation expression.

Return Value

Double value.

NULL if a value passed to the function is NULL.

Example

The following expression returns the tangent for all values in the Degrees port:

```
TAN( DEGREES * 3.14159 / 180 )
```

DEGREES	RETURN VALUE
70	2.74747741945531
50	1.19175359259435
30	0.577350269189672
5	0.0874886635259298
18	0.324919696232929
89	57.2899616310952
NULL	NULL

TANH

Availability:
Designer
Workflow Manager

Returns the hyperbolic tangent of the numeric value passed to this function.

Syntax

```
TANH( numeric_value )
```

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. Numeric data expressed in radians (degrees multiplied by pi divided by 180). Passes the numeric values for which you want to calculate the hyperbolic tangent. You can enter any valid transformation expression.

Return Value

Double value.

NULL if a value passed to the function is NULL.

Example

The following expression returns the hyperbolic tangent for the values in the Angles port:

```
TANH( ANGLES )
```

ANGLES	RETURN VALUE
1.0	0.761594155955765
2.897	0.993926947790665
3.66	0.998676551914886
5.45	0.999963084213409
0	0.0
0.345	0.331933853503641
NULL	NULL

Tip

You can perform arithmetic on the values passed to TANH before the function calculates the hyperbolic tangent. For example:

```
TANH( ARCS / 360 )
```

TO_BIGINT

Availability:
Designer
Workflow Manager

Converts a string or numeric value to a bigint value. TO_BIGINT syntax contains an optional argument that you can choose to round the number to the nearest integer or truncate the decimal portion. TO_BIGINT ignores leading blanks.

Syntax

```
TO_BIGINT( value [, flag] )
```

Argument	Required/ Optional	Description
value	Required	String or numeric datatype. Passes the value you want to convert to a bigint value. You can enter any valid transformation expression.
flag	Optional	Specifies whether to truncate or round the decimal portion. The flag must be an integer literal or the constants TRUE or FALSE. TO_BIGINT truncates the decimal portion when the flag is TRUE or a number other than 0. TO_BIGINT rounds the value to the nearest integer if the flag is FALSE or 0 or if you omit this argument. The flag is not set by default.

Return Value

Bigint.

NULL if the value passed to the function is NULL.

0 if the value passed to the function contains alphanumeric characters.

Examples

The following expressions use values from the port IN_TAX:

```
TO_BIGINT( IN_TAX, TRUE )
```

IN_TAX	RETURN VALUE
'7245176201123435.6789'	7245176201123435
'7245176201123435.2'	7245176201123435
'7245176201123435.2.48'	7245176201123435
NULL	NULL
'A12.3Grove'	0
' 176201123435.87'	176201123435
'-7245176201123435.2'	-7245176201123435
'-7245176201123435.23'	-7245176201123435
-9223372036854775806.9	-9223372036854775806
9223372036854775806.9	9223372036854775806

```
TO_BIGINT( IN_TAX )
```

IN_TAX	RETURN VALUE
'7245176201123435.6789'	7245176201123436
'7245176201123435.2'	7245176201123435
'7245176201123435.348'	7245176201123435
NULL	NULL
'A12.3Grove'	0
' 176201123435.87'	176201123436
'-7245176201123435.6789'	-7245176201123436
'-7245176201123435.23'	-7245176201123435
-9223372036854775806.9	-9223372036854775807
9223372036854775806.9	9223372036854775807

TO_CHAR (Dates)

Availability:
Designer
Workflow Manager

Converts dates to character strings. TO_CHAR also converts numeric values to strings. You can convert the date into any format using the TO_CHAR format strings.

Syntax

```
TO_CHAR( date [,format] )
```

Argument	Required/ Optional	Description
date	Required	Date/Time datatype. Passes the date values you want to convert to character strings. You can enter any valid transformation expression.
format	Optional	Enter a valid TO_CHAR format string. The format string defines the format of the return value, not the format for the values in the date argument. If you omit the format string, the function returns a string based on the date format specified in the session.

Return Value

String.

NULL if a value passed to the function is NULL.

Examples

The following expression converts the dates in the DATE_PROMISED port to text in the format MON DD YYYY:

```
TO_CHAR( DATE_PROMISED, 'MON DD YYYY' )
```

DATE_PROMISED	RETURN VALUE
Apr 1 1998 12:00:10AM	'Apr 01 1998'
Feb 22 1998 01:31:10PM	'Feb 22 1998'
Oct 24 1998 02:12:30PM	'Oct 24 1998'
NULL	NULL

If you omit the *format* argument, TO_CHAR returns a string in the date format specified in the session, by default, MM/DD/YYYY HH24:MI:SS.US:

```
TO_CHAR( DATE_PROMISED )
```

DATE_PROMISED	RETURN VALUE
Apr 1 1998 12:00:10AM	'04/01/1998 00:00:10.000000'
Feb 22 1998 01:31:10PM	'02/22/1998 13:31:10.000000'
Oct 24 1998 02:12:30PM	'10/24/1998 14:12:30.000000'
NULL	NULL

The following expressions return the day of the week for each date in a port:

```
TO_CHAR( DATE_PROMISED, 'D' )
```

DATE_PROMISED	RETURN VALUE
04-01-1997 12:00:10AM	'3'
02-22-1997 01:31:10PM	'7'

DATE_PROMISED	RETURN VALUE
10-24-1997 02:12:30PM	'6'
NULL	NULL

```
TO_CHAR( DATE_PROMISED, 'DAY' )
```

DATE_PROMISED	RETURN VALUE
04-01-1997 12:00:10AM	'Tuesday'
02-22-1997 01:31:10PM	'Saturday'
10-24-1997 02:12:30PM	'Friday'
NULL	NULL

The following expression returns the day of the month for each date in a port:

```
TO_CHAR( DATE_PROMISED, 'DD' )
```

DATE_PROMISED	RETURN VALUE
04-01-1997 12:00:10AM	'01'
02-22-1997 01:31:10PM	'22'
10-24-1997 02:12:30PM	'24'
NULL	NULL

The following expression returns the day of the year for each date in a port:

```
TO_CHAR( DATE_PROMISED, 'DDD' )
```

DATE_PROMISED	RETURN VALUE
04-01-1997 12:00:10AM	'091'
02-22-1997 01:31:10PM	'053'
10-24-1997 02:12:30PM	'297'
NULL	NULL

The following expressions return the hour of the day for each date in a port:

```
TO_CHAR( DATE_PROMISED, 'HH' )
TO_CHAR( DATE_PROMISED, 'HH12' )
```

DATE_PROMISED	RETURN VALUE
04-01-1997 12:00:10AM	'12'
02-22-1997 01:31:10PM	'01'
10-24-1997 02:12:30PM	'02'
NULL	NULL

```
TO_CHAR( DATE_PROMISED, 'HH24' )
```

DATE_PROMISED	RETURN VALUE
04-01-1997 12:00:10AM	'00'
02-22-1997 01:31:10PM	'13'
10-24-1997 11:12:30PM	'23'
NULL	NULL

The following expression converts date values to MJD values expressed as strings:

```
TO_CHAR( SHIP_DATE, 'J')
```

SHIP_DATE	RETURN_VALUE
Dec 31 1999 03:59:59PM	2451544
Jan 1 1900 01:02:03AM	2415021

The following expression converts dates to strings in the format MM/DD/YY:

```
TO_CHAR( SHIP_DATE, 'MM/DD/RR')
```

SHIP_DATE	RETURN_VALUE
12/31/1999 01:02:03AM	12/31/99
09/15/1996 03:59:59PM	09/15/96
05/17/2003 12:13:14AM	05/17/03

You can also use the format string SSSSS in a TO_CHAR expression. For example, the following expression converts the dates in the SHIP_DATE port to strings representing the total seconds since midnight:

```
TO_CHAR( SHIP_DATE, 'SSSSS')
```

SHIP_DATE	RETURN_VALUE
12/31/1999 01:02:03AM	3783
09/15/1996 03:59:59PM	86399

In TO_CHAR expressions, the YY format string produces the same results as the RR format string.

The following expression converts dates to strings in the format MM/DD/YY:

```
TO_CHAR( SHIP_DATE, 'MM/DD/YY')
```

SHIP_DATE	RETURN_VALUE
12/31/1999 01:02:03AM	12/31/99
09/15/1996 03:59:59PM	09/15/96
05/17/2003 12:13:14AM	05/17/03

The following expression returns the week of the month for each date in a port:

```
TO_CHAR( DATE_PROMISED, 'W' )
```

DATE_PROMISED	RETURN VALUE
04-01-1997 12:00:10AM	'01'
02-22-1997 01:31:10AM	'04'
10-24-1997 02:12:30PM	'04'
NULL	NULL

The following expression returns the week of the year for each date in a port:

```
TO_CHAR( DATE_PROMISED, 'WW' )
```

DATE_PROMISED	RETURN VALUE
04-01-1997 12:00:10PM	'18'
02-22-1997 01:31:10AM	'08'
10-24-1997 02:12:30AM	'43'
NULL	NULL

Tip

You can combine TO_CHAR and TO_DATE to convert a numeric value for a month into the text value for a month using a function such as:

```
TO_CHAR( TO_DATE( numeric_month, 'MM' ), 'MONTH' )
```

TO_CHAR (Numbers)

Availability:
Designer
Workflow Manager

Converts numeric values to text strings. TO_CHAR also converts dates to strings.

TO_CHAR converts numeric values to text strings as follows:

- Converts double values to strings of up to 16 digits and provides accuracy up to 15 digits. If you pass a number with more than 15 digits, TO_CHAR rounds the number to the sixteenth digit.
- Returns decimal notation for numbers in the ranges (-1e16,-1e-16] and [1e-16, 1e16). TO_CHAR returns scientific notation for numbers outside these ranges.

Note: The Integration Service converts the values 1e-16 and -1e16 to scientific notation, but returns the values 1e-16 and -1e-16 in decimal notation.

Syntax

```
TO_CHAR( numeric_value )
```

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. The numeric value you want to convert to a string. You can enter any valid transformation expression.

Return Value

String.

NULL if a value passed to the function is NULL.

Example

The following expression converts the values in the SALES port to text:

```
TO_CHAR( SALES )
```

```
SALES                          RETURN VALUE
1010.99                        '1010.99'
-15.62567                      '-15.62567'
10842764968208837340           '1.084276496820884e+019'  (rounded to 16th digit)
1.234567890123456789e-10       '0.00000000012345678901234567'
                               (greater than 1e-16 but less than 1e16)
1.23456789012345e17            '1.23456789012345e17'  (greater than 1e16)
0                              '0'
33.15                          '33.15'
NULL                           NULL
```

TO_DATE

Availability:
Designer
Workflow Manager

Converts a character string to a Date/Time datatype. You use the TO_DATE format strings to specify the format of the source strings.

The output port must be Date/Time for TO_DATE expressions.

If you are converting two-digit years with TO_DATE, use either the RR or YY format string. Do not use the YYYY format string.

Syntax

```
TO_DATE( string [, format] )
```

Argument	Required/ Optional	Description
string	Required	Must be a string datatype. Passes the values that you want to convert to dates. You can enter any valid transformation expression.
format	Optional	Enter a valid TO_DATE format string. The format string must match the parts of the *string* argument. For example, if you pass the string 'Mar 15 1998 12:43:10AM', you must use the format string 'MON DD YYYY HH12:MI:SSAM'. If you omit the format string, the string value must be in the date format specified in the session.

Return Value

Date.

TO_DATE always returns a date and time. If you pass a string that does not have a time value, the date returned always includes the time 00:00:00.000000000. You can map the results of this function to any target column with a datetime datatype. If the target column precision is less than nanoseconds, the Integration Service truncates the datetime value to match the precision of the target column when it writes datetime values to the target.

NULL if you pass a null value to this function.

Warning: The format of the TO_DATE string must match the format string including any date separators. If it does not, the Integration Service might return inaccurate values or skip the record.

Examples

The following expression returns date values for the strings in the DATE_PROMISED port. TO_DATE always returns a date and time. If you pass a string that does not have a time value, the date returned always includes the time 00:00:00.000000000. If you run a session in the twentieth century, the century will be 19. In this example, the current year on the node running the Integration Service is 1998. The datetime format for the target column is MON DD YY HH24:MI SS, so the Integration Service truncates the datetime value to seconds when it writes to the target:

```
TO_DATE( DATE_PROMISED, 'MM/DD/YY' )
```

```
DATE_PROMISED                    RETURN VALUE
'01/22/98'                       Jan 22 1998 00:00:00
'05/03/98'                       May 3 1998 00:00:00
'11/10/98'                       Nov 10 1998 00:00:00
```

DATE_PROMISED	RETURN VALUE
'10/19/98'	Oct 19 1998 00:00:00
NULL	NULL

The following expression returns date and time values for the strings in the DATE_PROMISED port. If you pass a string that does not have a time value, the Integration Service returns an error. If you run a session in the twentieth century, the century will be 19. The current year on the node running the Integration Service is 1998:

```
TO_DATE( DATE_PROMISED, 'MON DD YYYY HH12:MI:SSAM' )
```

DATE_PROMISED	RETURN VALUE
'Jan 22 1998 02:14:56PM'	Jan 22 1998 02:14:56PM
'Mar 15 1998 11:11:11AM'	Mar 15 1998 11:11:11AM
'Jun 18 1998 10:10:10PM'	Jun 18 1998 10:10:10PM
'October 19 1998'	*Error. Integration Service skips this row.*
NULL	NULL

The following expression converts strings in the SHIP_DATE_MJD_STRING port to date values:

```
TO_DATE (SHIP_DATE_MJD_STR, 'J')
```

SHIP_DATE_MJD_STR	RETURN_VALUE
'2451544'	Dec 31 1999 00:00:00.000000000
'2415021'	Jan 1 1900 00:00:00.000000000

Because the J format string does not include the time portion of a date, the return values have the time set to 00:00:00.000000000.

The following expression converts a string to a four-digit year format. The current year is 1998:

```
TO_DATE( DATE_STR, 'MM/DD/RR')
```

DATE_STR	RETURN VALUE
'04/01/98'	04/01/1998 00:00:00.000000000
'08/17/05'	08/17/2005 00:00:00.000000000

The following expression converts a string to a four-digit year format. The current year is 1998:

```
TO_DATE( DATE_STR, 'MM/DD/YY')
```

DATE_STR	RETURN VALUE
'04/01/98'	04/01/1998 00:00:00.000000000
'08/17/05'	08/17/1905 00:00:00.000000000

Note: For the second row, RR returns the year 2005 and YY returns the year 1905.

The following expression converts a string to a four-digit year format. The current year is 1998:

```
TO_DATE( DATE_STR, 'MM/DD/Y')
```

DATE_STR	RETURN VALUE
'04/01/8'	04/01/1998 00:00:00.000000000
'08/17/5'	08/17/1995 00:00:00.000000000

The following expression converts a string to a four-digit year format. The current year is 1998:

```
TO_DATE( DATE_STR, 'MM/DD/YYY')
```

DATE_STR	RETURN VALUE
'04/01/998'	04/01/1998 00:00:00.000000000
'08/17/995'	08/17/1995 00:00:00.000000000

The following expression converts strings that includes the seconds since midnight to date values:

```
TO_DATE( DATE_STR, 'MM/DD/YYYY SSSSS')
```

DATE_STR	RETURN_VALUE
'12/31/1999 3783'	12/31/1999 01:02:03
'09/15/1996 86399'	09/15/1996 23:59:59

If the target accepts different date formats, use TO_DATE and IS_DATE with the DECODE function to test for acceptable formats. For example:

```
DECODE( TRUE,

 --test first format
   IS_DATE( CLOSE_DATE,'MM/DD/YYYY HH24:MI:SS' ),

 --if true, convert to date
      TO_DATE( CLOSE_DATE,'MM/DD/YYYY HH24:MI:SS' ),

 --test second format; if true, convert to date
   IS_DATE( CLOSE_DATE,'MM/DD/YYYY'), TO_DATE( CLOSE_DATE,'MM/DD/YYYY' ),

 --test third format; if true, convert to date
   IS_DATE( CLOSE_DATE,'MON DD YYYY'), TO_DATE( CLOSE_DATE,'MON DD YYYY'),

 --if none of the above
   ERROR( 'NOT A VALID DATE') )
```

You can combine TO_CHAR and TO_DATE to convert a numeric value for a month into the text value for a month using a function such as:

```
TO_CHAR( TO_DATE( numeric_month, 'MM' ), 'MONTH' )
```

Related Topics:

- "Rules and Guidelines for Date Format Strings" on page 32

TO_DECIMAL

Availability:
Designer

Converts a string or numeric value to a decimal value. TO_DECIMAL ignores leading blanks.

Syntax

```
TO_DECIMAL( value [, scale] )
```

Argument	Required/ Optional	Description
value	Required	Must be a string or numeric datatype. Passes the values you want to convert to decimals. You can enter any valid transformation expression.
scale	Optional	Must be an integer literal between 0 and 28, inclusive. Specifies the number of digits allowed after the decimal point. If you omit this argument, the function returns a value with the same scale as the input value.

Return Value

If the string contains a non-numeric character, converts the numeric portion of the string up to the first non-numeric character.

If the first numeric character is non-numeric, returns 0.

Decimal of precision and scale between 0 and 28, inclusive.

NULL if a value passed to the function is NULL.

Note: If the return value is decimal with precision greater than 15, you can enable high precision in the session properties to ensure decimal precision up to 28 digits.

Example

This expression uses values from the port IN_TAX. The datatype is decimal with precision of 10 and scale of 3:

```
TO_DECIMAL( IN_TAX, 3 )
```

IN_TAX	RETURN VALUE
'15.6789'	15.679
'60.2'	60.200
'118.348'	118.348
NULL	NULL
'A12.3Grove'	0
'711A1'	711

TO_FLOAT

Availability:
Designer
Workflow Manager

Converts a string or numeric value to a double-precision floating point number (the Double datatype). TO_FLOAT ignores leading blanks.

Syntax

```
TO_FLOAT( value )
```

Argument	Required/ Optional	Description
value	Required	Must be a string or numeric datatype. Passes the values you want to convert to double values. You can enter any valid transformation expression.

Return Value

Double value.

0 if the value in the port is blank or a non-numeric character.

NULL if a value passed to this function is NULL.

Example

This expression uses values from the port IN_TAX:

```
TO_FLOAT( IN_TAX )
```

IN_TAX	RETURN VALUE
'15.6789'	15.6789
'60.2'	60.2
'118.348'	118.348
NULL	NULL
'A12.3Grove'	0

TO_INTEGER

Availability:
Designer
Workflow Manager

Converts a string or numeric value to an integer. TO_INTEGER syntax contains an optional argument that you can choose to round the number to the nearest integer or truncate the decimal portion. TO_INTEGER ignores leading blanks.

Syntax

```
TO_INTEGER( value [, flag] )
```

Argument	Required/ Optional	Description
value	Required	String or numeric datatype. Passes the value you want to convert to an integer. You can enter any valid transformation expression.
flag	Optional	Specifies whether to truncate or round the decimal portion. The flag must be an integer literal or the constants TRUE or FALSE. TO_INTEGER truncates the decimal portion when the flag is TRUE or a number other than 0. TO_INTEGER rounds the value to the nearest integer if the flag is FALSE or 0 or if you omit this argument.

Return Value

Integer.

NULL if the value passed to the function is NULL.

0 if the value passed to the function contains alphanumeric characters.

Examples

The following expressions use values from the port IN_TAX. The Integration Service displays an error when the conversion causes a numeric overflow:

```
TO_INTEGER( IN_TAX, TRUE )
```

IN_TAX	RETURN VALUE
'15.6789'	15

IN_TAX	RETURN VALUE
'60.2'	60
'118.348'	118
‘5,000,000,000’	*Error. Integration Service doesn't write row.*
NULL	NULL
'A12.3Grove'	0
' 123.87'	123
'-15.6789'	-15
'-15.23'	-15

```
TO_INTEGER( IN_TAX, FALSE)
```

IN_TAX	RETURN VALUE
'15.6789'	16
'60.2'	60
'118.348'	118
‘5,000,000,000’	*Error. Integration Service doesn't write row.*
NULL	NULL
'A12.3Grove'	0
' 123.87'	124
'-15.6789'	-16
'-15.23'	-15

TRUNC (Dates)

Availability:
Designer
Workflow Manager

Truncates dates to a specific year, month, day, hour, minute, second, millisecond, or microsecond. You can also use TRUNC to truncate numbers.

You can truncate the following date parts:

- **Year.** If you truncate the year portion of the date, the function returns Jan 1 of the input year with the time set to 00:00:00.000000000. For example, the following expression returns 1/1/1997 00:00:00.000000000:

  ```
  TRUNC(12/1/1997 3:10:15, 'YY')
  ```

- **Month.** If you truncate the month portion of a date, the function returns the first day of the month with the time set to 00:00:00.000000000. For example, the following expression returns 4/1/1997 00:00:00.000000000:

  ```
  TRUNC(4/15/1997 12:15:00, 'MM')
  ```

- **Day.** If you truncate the day portion of a date, the function returns the date with the time set to 00:00:00.000000000. For example, the following expression returns 6/13/1997 00:00:00.000000000:

  ```
  TRUNC(6/13/1997 2:30:45, 'DD')
  ```

- **Hour.** If you truncate the hour portion of a date, the function returns the date with the minutes, seconds, and subseconds set to 0. For example, the following expression returns 4/1/1997 11:00:00.000000000:

  ```
  TRUNC(4/1/1997 11:29:35, 'HH')
  ```

- **Minute.** If you truncate the minute portion of a date, the function returns the date with the seconds and subseconds set to 0. For example, the following expression returns 5/22/1997 10:15:00.000000000:

  ```
  TRUNC(5/22/1997 10:15:29, 'MI')
  ```

- **Second.** If you truncate the second portion of a date, the function returns the date with the milliseconds set to 0. For example, the following expression returns 5/22/1997 10:15:29.000000000:

```
TRUNC(5/22/1997 10:15:29.135, 'SS')
```

- **Millisecond.** If you truncate the millisecond portion of a date, the function returns the date with the microseconds set to 0. For example, the following expression returns 5/22/1997 10:15:30.135000000:

```
TRUNC(5/22/1997 10:15:30.135235, 'MS')
```

- **Microsecond.** If you truncate the microsecond portion of a date, the function returns the date with the nanoseconds set to 0. For example, the following expression returns 5/22/1997 10:15:30.135235000:

```
TRUNC(5/22/1997 10:15:29.135235478, 'US')
```

Syntax

```
TRUNC( date [,format] )
```

Argument	Required/ Optional	Description
date	Required	Date/Time datatype. The date values you want to truncate. You can enter any valid transformation expression that evaluates to a date.
format	Optional	Enter a valid format string. The format string is not case sensitive. If you omit the format string, the function truncates the time portion of the date, setting it to 00:00:00.000000000.

Return Value

Date.

NULL if a value passed to the function is NULL.

Examples

The following expressions truncate the year portion of dates in the DATE_SHIPPED port:

```
TRUNC( DATE_SHIPPED, 'Y' )
TRUNC( DATE_SHIPPED, 'YY' )
TRUNC( DATE_SHIPPED, 'YYY' )
TRUNC( DATE_SHIPPED, 'YYYY' )
```

DATE_SHIPPED	RETURN VALUE
Jan 15 1998 2:10:30AM	Jan 1 1998 12:00:00.000000000
Apr 19 1998 1:31:20PM	Jan 1 1998 12:00:00.000000000
Jun 20 1998 3:50:04AM	Jan 1 1998 12:00:00.000000000
Dec 20 1998 3:29:55PM	Jan 1 1998 12:00:00.000000000
NULL	NULL

The following expressions truncate the month portion of each date in the DATE_SHIPPED port:

```
TRUNC( DATE_SHIPPED, 'MM' )
TRUNC( DATE_SHIPPED, 'MON' )
TRUNC( DATE_SHIPPED, 'MONTH' )
```

DATE_SHIPPED	RETURN VALUE
Jan 15 1998 2:10:30AM	Jan 1 1998 12:00:00.000000000AM
Apr 19 1998 1:31:20PM	Apr 1 1998 12:00:00.000000000AM
Jun 20 1998 3:50:04AM	Jun 1 1998 12:00:00.000000000AM
Dec 20 1998 3:29:55PM	Dec 1 1998 12:00:00.000000000AM
NULL	NULL

The following expressions truncate the day portion of each date in the DATE_SHIPPED port:

```
TRUNC( DATE_SHIPPED, 'D' )
TRUNC( DATE_SHIPPED, 'DD' )
```

```
TRUNC( DATE_SHIPPED, 'DDD' )
TRUNC( DATE_SHIPPED, 'DY' )
TRUNC( DATE_SHIPPED, 'DAY' )
```

DATE_SHIPPED	RETURN VALUE
Jan 15 1998 2:10:30AM	Jan 15 1998 12:00:00.000000000AM
Apr 19 1998 1:31:20PM	Apr 19 1998 12:00:00.000000000AM
Jun 20 1998 3:50:04AM	Jun 20 1998 12:00:00.000000000AM
Dec 20 1998 3:29:55PM	Dec 20 1998 12:00:00.000000000AM
Dec 31 1998 11:59:59PM	Dec 31 1998 12:00:00.000000000AM
NULL	NULL

The following expressions truncate the hour portion of each date in the DATE_SHIPPED port:

```
TRUNC( DATE_SHIPPED, 'HH' )
TRUNC( DATE_SHIPPED, 'HH12' )
TRUNC( DATE_SHIPPED, 'HH24' )
```

DATE_SHIPPED	RETURN VALUE
Jan 15 1998 2:10:31AM	Jan 15 1998 2:00:00.000000000AM
Apr 19 1998 1:31:20PM	Apr 19 1998 1:00:00.000000000PM
Jun 20 1998 3:50:04AM	Jun 20 1998 3:00:00.000000000AM
Dec 20 1998 3:29:55PM	Dec 20 1998 3:00:00.000000000PM
Dec 31 1998 11:59:59PM	Dec 31 1998 11:00:00.000000000AM
NULL	NULL

The following expression truncates the minute portion of each date in the DATE_SHIPPED port:

```
TRUNC( DATE_SHIPPED, 'MI' )
```

DATE_SHIPPED	RETURN VALUE
Jan 15 1998 2:10:30AM	Jan 15 1998 2:10:00.000000000AM
Apr 19 1998 1:31:20PM	Apr 19 1998 1:31:00.000000000PM
Jun 20 1998 3:50:04AM	Jun 20 1998 3:50:00.000000000AM
Dec 20 1998 3:29:55PM	Dec 20 1998 3:29:00.000000000PM
Dec 31 1998 11:59:59PM	Dec 31 1998 11:59:00.000000000PM
NULL	NULL

TRUNC (Numbers)

Availability:
Designer
Workflow Manager

Truncates numbers to a specific digit. You can also use TRUNC to truncate dates.

Syntax

```
TRUNC( numeric_value [, precision] )
```

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. Passes the values you want to truncate. You can enter any valid transformation expression that evaluates to a Numeric datatype.
precision	Optional	Can be a positive or negative integer. You can enter any valid transformation expression that evaluates to an integer. The integer specifies the number of digits to truncate.

If *precision* is a positive integer, TRUNC returns *numeric_value* with the number of decimal places specified by *precision*. If *precision* is a negative integer, TRUNC changes the specified digits to the left of the decimal point to zeros. If you omit the *precision* argument, TRUNC truncates the decimal portion of *numeric_value* and returns an integer.

If you pass a decimal *precision* value, the Integration Service rounds *numeric_value* to the nearest integer before evaluating the expression.

Return Value

Numeric value.

NULL if one of the arguments is NULL.

Note: If the return value is decimal with precision greater than 15, you can enable high precision in the session properties to ensure decimal precision up to 28 digits.

Examples

The following expressions truncate the values in the Price port:

```
TRUNC( PRICE, 3 )
```

PRICE	RETURN VALUE
12.9995	12.999
-18.8652	-18.865
56.9563	56.956
15.9928	15.992
NULL	NULL

```
TRUNC( PRICE, -1 )
```

PRICE	RETURN VALUE
12.99	10.0
-187.86	-180.0
56.95	50.0
1235.99	1230.0
NULL	NULL

```
TRUNC( PRICE )
```

PRICE	RETURN VALUE
12.99	12.0
-18.99	-18.0
56.95	56.0
15.99	15.0
NULL	NULL

UPPER

Availability:
Designer
Workflow Manager

Converts lowercase string characters to uppercase.

Syntax

```
UPPER( string )
```

Argument	Required/ Optional	Description
string	Required	String datatype. Passes the values you want to change to uppercase text. You can enter any valid transformation expression.

Return Value

Uppercase string. If the data contains multibyte characters, the return value depends on the code page and data movement mode of the Integration Service.

NULL if a value passed to the function is NULL.

Example

The following expression changes all names in the FIRST_NAME port to uppercase:

```
UPPER( FIRST_NAME )
```

FIRST_NAME	RETURN VALUE
Ramona	RAMONA
NULL	NULL
THOMAS	THOMAS
PierRe	PIERRE
Bernice	BERNICE

VARIANCE

Availability:
Designer

Returns the variance of a value you pass to it. VARIANCE is used to analyze statistical data. You can nest only one other aggregate function within VARIANCE, and the nested function must return a Numeric datatype.

Syntax

```
VARIANCE( numeric_value [, filter_condition ] )
```

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. Passes the values for which you want to calculate a variance. You can enter any valid transformation expression.
filter_condition	Optional	Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression.

Formula

The function uses the following formula to calculate the variance:

$$\frac{\sum_{i=1}^{n} x_i^2 - \frac{1}{n}\left[\sum_{i=1}^{n} x_i\right]^2}{n-1}$$

Use the following guidelines for this formula:

- *xi* is one of the numeric values.
- *n* is the number of elements in the set of numeric values. If n is 1, the variance is 0.

Return Value

Double value.

NULL if all values passed to the function are NULL or if no rows are selected (for example, the *filter_condition* evaluates to FALSE or NULL for all rows).

Nulls

If a single value is NULL, VARIANCE ignores it. However, if all values passed to the function are NULL or if no rows are selected, VARIANCE returns NULL.

Note: By default, the Integration Service treats null values as NULLs in aggregate functions. If you pass an entire port or group of null values, the function returns NULL. However, when you configure the Integration Service, you can choose how you want to handle null values in aggregate functions. You can treat null values as 0 in aggregate functions or as NULL.

Group By

VARIANCE groups values based on group by ports you define in the transformation, returning one result for each group.

If there is not a group by port, VARIANCE treats all rows as one group, returning one value.

Example

The following expression calculates the variance of all rows in the TOTAL_SALES port:

```
VARIANCE( TOTAL_SALES )
```

```
TOTAL_SALES
2198.0
2256.0
3001.0
NULL
```

```
TOTAL_SALES
8953.0
RETURN VALUE:  10592444.6666667
```

CHAPTER 7

Creating Custom Functions

This chapter includes the following topics:

Creating Custom Functions Overview

Custom functions extend the library of PowerCenter functions. They are functions you create to use in transformation and workflow expressions.

You create custom functions outside of PowerCenter with the Custom Functions API. The Custom Functions API uses the C programming language. You can share custom functions with others. Users can add the functions to their repository and use them like a PowerCenter transformation language function.

This chapter includes a sample function that demonstrates how to create and use a custom function. The steps in this chapter create the ECHO function. This function takes an argument as input and returns the input value to the user. The sample code for the ECHO function is in the server\samples\exprsdk\echo directory.

You can also view a more complex sample custom function. The SampleLoanPayment custom function contains functions that are not available using C. SampleLoanPayment is in the server\samples\exprsdk\loanpayment directory.

Steps to Create Custom Functions

Complete the following steps to create custom functions:

1. **Get repository ID attributes.** Get repository ID attributes to include in the repository plug-in.
2. **Create the header file.** Define one or more custom functions in the header file.
3. **Create the implementation file.** Define one or more custom functions in the implementation file.
4. **Build the modules.** Build modules to create DLLs and shared libraries.

5. **Create the repository plug-in file.** Define metadata for custom functions.
6. **Test the custom functions.** Install custom functions and use them in a mapping and workflow for verification.

Installing Custom Functions

To use custom functions, you must add the functions to the PowerCenter environment.

Related Topics:

- "Installing Custom Functions" on page 193

Step 1. Get Repository ID Attributes

Before you develop a custom function, you must determine the repository ID attributes for the custom function repository plug-in. Use the repository ID attributes to identify the plug-in when you define the plug-in metadata.

To get repository ID attributes, perform one of the following tasks:

- **If you are distributing custom functions outside your organization, contact Informatica.** Informatica assigns each plug-in with unique repository ID attributes. Repository ID attributes are invalid if they conflict with those of another vendor. To obtain repository ID attributes, email Informatica at devnet@informatica.com.
- **If you only using custom functions within your organization, define repository ID attributes without contacting Informatica.** If you develop a plug-in for your organization that will be used with other plug-ins in PowerCenter, assign unique values to the repository ID attributes for each plug-in.

The following table shows the XML attributes that require unique values to define a plug-in:

Repository ID Attribute	Description
Plugin ID	Identifies the ID of the plug-in. This value corresponds to the ID attribute for the PLUGIN element.
Vendor ID	Identifies the vendor that develops the plug-in. This value corresponds to the VENDORID attribute for the PLUGIN element.
Function Group ID	Identifies the ID for the function group. This value corresponds to the ID attribute for the FUNCTION_GROUP element.
Function ID	Identifies the ID of the function. This value corresponds to the ID attribute for FUNCTION element.

Note: Repository ID attributes are invalid if they conflict with each other.

Related Topics:

- "The PLUGIN Element" on page 189
- "The FUNCTION_GROUP Element" on page 190
- "The FUNCTION Element" on page 191

Step 2. Create a Header File

Create a header file using C to declare all functions. Use one header file for one or more custom functions.

The following example shows the echo.h header file for the ECHO custom function:

```
#ifndef __ECHO_PLUGIN_HPP
#define __ECHO_PLUGIN_HPP

#if defined(WIN32)
    #if defined SAMPLE_EXPR_EXPORTS
        #define  SAMPLE_EXPR_SPEC __declspec(dllexport)
    #else
        #define  SAMPLE_EXPR_SPEC __declspec(dllimport)
    #endif
#else
    #define SAMPLE_EXPR_SPEC
#endif

// method to get description of Echo function
extern "C" SAMPLE_EXPR_SPEC IUNICHAR * getDescriptionEcho(IUNICHAR* ns, IUNICHAR* sFuncName);

// method to get prototype of Echo function
extern "C" SAMPLE_EXPR_SPEC IUNICHAR * getPrototypeEcho(IUNICHAR* ns, IUNICHAR* sFuncName);

// method to validate usage of Echo function
extern "C" SAMPLE_EXPR_SPEC INFA_EXPR_STATUS validateFunctionEcho(IUNICHAR* ns, IUNICHAR* sFuncName,
                              IUINT32 numArgs, INFA_EXPR_OPD_METADATA** inputArgList,
                              INFA_EXPR_OPD_METADATA* retValue);

// method to process row for Echo function
extern "C" SAMPLE_EXPR_SPEC INFA_EXPR_ROWSTATUS processRowEcho(INFA_EXPR_FUNCTION_INSTANCE_HANDLE
*fnInstance, IUNICHAR **errMsg);

// method to do module level initialization for Echo function
extern "C" SAMPLE_EXPR_SPEC INFA_EXPR_STATUS moduleInitEcho(INFA_EXPR_MODULE_HANDLE *modHandle);

// method to do module level deinitialization for Echo function
extern "C" SAMPLE_EXPR_SPEC INFA_EXPR_STATUS moduleDeinitEcho(INFA_EXPR_MODULE_HANDLE *modHandle);

// method to do function level initialization for Echo function
extern "C" SAMPLE_EXPR_SPEC INFA_EXPR_STATUS functionInitEcho(INFA_EXPR_FUNCTION_HANDLE *funHandle);

// method to do function level deinitialization for Echo function
extern "C" SAMPLE_EXPR_SPEC INFA_EXPR_STATUS functionDeinitEcho(INFA_EXPR_FUNCTION_HANDLE *funHandle);

// method to do function instance level initialization for Echo function
extern "C" SAMPLE_EXPR_SPEC INFA_EXPR_STATUS functionInstInitEcho(INFA_EXPR_FUNCTION_INSTANCE_HANDLE
*funInstHandle);

// method to do function instance level deinitialization for Echo function
extern "C" SAMPLE_EXPR_SPEC INFA_EXPR_STATUS functionInstDeinitEcho(INFA_EXPR_FUNCTION_INSTANCE_HANDLE
*funInstHandle);

/**
    These are all plugin callbacks, which have been implemented to get various module,
    function level interfaces
*/
// method to get plugin version
extern "C" SAMPLE_EXPR_SPEC INFA_EXPR_STATUS INFA_EXPR_GetPluginVersion(INFA_VERSION* sdkVersion,
INFA_VERSION* pluginVersion);

// method to delete the string allocated by this plugin. used for deleting the error
// messages
extern "C" SAMPLE_EXPR_SPEC void INFA_EXPR_DestroyString(IUNICHAR *);

// method to get validation interfaces
extern "C" SAMPLE_EXPR_SPEC INFA_EXPR_STATUS INFA_EXPR_ValidateGetUserInterface( IUNICHAR* ns,
IUNICHAR* sFuncName, INFA_EXPR_VALIDATE_METHODS* functions);

// method to get module interfaces
extern "C" SAMPLE_EXPR_SPEC INFA_EXPR_STATUS INFA_EXPR_ModuleGetUserInterface(INFA_EXPR_LIB_METHODS*
functions);
```

```
// method to get function interfaces
extern "C" SAMPLE_EXPR_SPEC INFA_EXPR_STATUS INFA_EXPR_FunctionGetUserInterface(IUNICHAR*
nameSpaceName, IUNICHAR* functionName, INFA_EXPR_FUNCTION_METHODS* functions);

// method to get function instance interfaces
extern "C" SAMPLE_EXPR_SPEC INFA_EXPR_STATUS INFA_EXPR_FunctionInstanceGetUserInterface(IUNICHAR*
nameSpaceName, IUNICHAR* functionName, INFA_EXPR_FUNCTION_INSTANCE_METHODS* functions);

#endif
```

Step 3. Create an Implementation File

The implementation file contains the definitions of the functions you use to create a custom function. Create an implementation file using C. You can use one implementation file for one or more custom functions. You can also use one implementation file to define both the validation and runtime functions of a custom function.

The following example shows the echo.c implementation file for the ECHO custom function:

```
/****************************************************************************
 *
 * Copyright (c) 2005 Informatica Corporation.  This file contains
 * material proprietary to Informatica Corporation and may not be copied
 * or distributed in any form without the written permission of Informatica
 * Corporation.
 *
 ****************************************************************************/

/****************************************************************************
 * ECHO function Procedure File
 *
 * This file contains code that creates the ECHO function, which the
 * Integration Service calls during a workflow.
 ****************************************************************************/

/* Informatica ECHO function example developed using the Custom Function
 * API.

 * Filename: Echo.c

 * An example of a custom function developed using PowerCenter
 *
 * The purpose of the ECHO function is to return the input value to the user.
 *
 */

/****************************************************************************
                    Includes
 ****************************************************************************/
#include <stdio.h>
#include <string.h>
#include "sdkexpr/exprsdk.h"

#define SAMPLE_EXPR_EXPORTS
#include "SampleExprPlugin.hpp"


static IUNICHAR ECHO_STR[80];


/****************************************************************************
                    Functions
 ****************************************************************************/

/****************************************************************************
    Function: INFA_EXPR_GetPluginVersion

Description: Defines the version of the plug-in. It must be the same as the
Custom Function API version. Returns ISUCCESS if the plug-in version
matches the Custom Function API version.  Otherwise, returns IFAILURE.
```

```
Input:  sdkVersion - Current version of the Custom Function API.
Output: pluginVersion - Set the version of the plug-in.
Remarks: Custom Function API checks for compatibility between itself and the
plug-in version.
 ************************************************************************/

 extern "C" SAMPLE_EXPR_SPEC
  INFA_EXPR_STATUS INFA_EXPR_GetPluginVersion(INFA_VERSION* sdkVersion, INFA_VERSION* pluginVersion)
{
    pluginVersion->m_major = 1;
    pluginVersion->m_minor = 0;
    pluginVersion->m_patch = 0;

    INFA_EXPR_STATUS retStatus;
    retStatus.status = ISUCCESS;
    retStatus.errMsg = NULL;
    return retStatus;
}

/************************************************************************
    Function: INFA_EXPR_DestroyString

Description: Destroys all strings the plug-in returns. For example, it
destroys error messages or the return value of other function calls, such
as getFunctionDescription. Returns no value.

Input:  The pointer to the allocated string.
Output: N/A
Remarks: Frees the memory to avoid issues with multiple heaps.
 ************************************************************************/

extern "C" SAMPLE_EXPR_SPEC void INFA_EXPR_DestroyString(IUNICHAR *strToDelete)
{
    delete [] strToDelete;
}


/************************************************************************
    Function: INFA_EXPR_ValidateGetUserInterface

Description: Returns function pointers to the validation functions. Returns
ISUCCESS when the plug-in implemented the function. Returns IFAILURE
when the plug-in did not implement the function or another error occurred.

Input: Namespace and name of function.
Output: Functions. The plug-in needs to set various function pointers.
Remarks: Check the namespace and function name for validaity. Set the various
function pointers appropriately.
 ************************************************************************/

extern "C" SAMPLE_EXPR_SPEC
    INFA_EXPR_STATUS INFA_EXPR_ValidateGetUserInterface(IUNICHAR* ns, IUNICHAR* sFuncName,
                                                INFA_EXPR_VALIDATE_METHODS* functions)
{
    INFA_EXPR_STATUS retStatus;
    retStatus.errMsg = NULL;

    // check function name is not null
    if (!sFuncName)
    {
        retStatus.status = IFAILURE;
        return retStatus;
    }

    // set the appropriate function pointers
    functions->validateFunction = validateFunctionEcho;
    functions->getFunctionDescription = getDescriptionEcho;
    functions->getFunctionPrototype = getPrototypeEcho;

    retStatus.status = ISUCCESS;
    return retStatus;
}

/************************************************************************
    Function: INFA_EXPR_ModuleGetUserInterface
```

```
Description: Sets the function pointers for module-level interaction.
Returns ISUCCESS when functions pointers are set appropriately. Otherwise,
returns IFAILURE.

Input: N/A
Output: Functions. The plug-in needs to set various function pointers.
Remarks: Set the module init/deinit function pointers.
 ****************************************************************************/

extern "C" SAMPLE_EXPR_SPEC
    INFA_EXPR_STATUS INFA_EXPR_ModuleGetUserInterface(INFA_EXPR_LIB_METHODS* functions)
{
    functions->module_init = moduleInitEcho;
    functions->module_deinit = moduleDeinitEcho;

    INFA_EXPR_STATUS retStatus;
    retStatus.status = ISUCCESS;
    retStatus.errMsg = NULL;
    return retStatus;
}

/****************************************************************************
    Function: INFA_EXPR_FunctionGetUserInterface

Description: Sets the function pointers for function-level interaction.
PowerCenter calls this function for every custom function this library
implements. Returns ISUCCESS when The plugin implements this function and
sets the function pointers correctly. Otherwise, returns IFAILURE.

Input:  Namespace and name of function.
Output: Functions. The plug-in needs to set function pointers for function
init/deinit.
Remarks: Set the function init/deinit function pointers.
 ****************************************************************************/

extern "C" SAMPLE_EXPR_SPEC
    INFA_EXPR_STATUS INFA_EXPR_FunctionGetUserInterface(IUNICHAR* nameSpaceName,
                                                        IUNICHAR* functionName,
                                                        INFA_EXPR_FUNCTION_METHODS* functions)
{
    functions->function_init = functionInitEcho;
    functions->function_deinit = functionDeinitEcho;

    INFA_EXPR_STATUS retStatus;
    retStatus.status = ISUCCESS;
    retStatus.errMsg = NULL;
    return retStatus;
}

/****************************************************************************
    Function: INFA_EXPR_FunctionInstanceGetUserInterface

Description: Sets the function pointers for function instance-level
interaction. PowerCenter calls this function for every custom function this
library implements. Returns ISUCCESS when The plugin implements this
function and sets the function pointers correctly. Otherwise, returns
IFAILURE.

Input:  Namespace and name of function.
Output: Functions. The plug-in needs to set function pointers for instance
init/deinit/processrow.
Remarks: Set the function instance init/deinit/processrow function pointers.
 ****************************************************************************/

extern "C" SAMPLE_EXPR_SPEC
    INFA_EXPR_STATUS INFA_EXPR_FunctionInstanceGetUserInterface(IUNICHAR* nameSpaceName,
                                                                IUNICHAR* functionName,
                                                                INFA_EXPR_FUNCTION_INSTANCE_METHODS* functions)
{
    functions->fnInstance_init = functionInstInitEcho;
    functions->fnInstance_processRow = processRowEcho;
    functions->fnInstance_deinit = functionInstDeinitEcho;

    INFA_EXPR_STATUS retStatus;
    retStatus.status = ISUCCESS;
```

```
    retStatus.errMsg = NULL;
    return retStatus;
}

/*************************************************************************
    Function: INFA_EXPR_getDescriptionEcho

Description: Gets the description of the ECHO function. It calls
destroyString to delete the arguments from memory when usage is complete.
The return value must be a null-terminated string.

Input:  Namespace and name of function.
Output: Description of the function.
Remarks: Returns the description of function. The Custom Functions API calls
destroy string to free the allocated memory.
 *************************************************************************/

extern "C" SAMPLE_EXPR_SPEC
    IUNICHAR * getDescriptionEcho(IUNICHAR* ns, IUNICHAR* sFuncName)
{
    static IUNICHAR *uniDesc = NULL;
    const char *description = "Echoes the input";

    if (uniDesc)
        return uniDesc;

    int i, len;
    len = strlen(description);
    uniDesc = new IUNICHAR[2*len+2];
    for (i=0; i<len; i++)
    {
        uniDesc[i] = description[i];
    }
    uniDesc[i] = 0;
    return uniDesc;
}

/*************************************************************************
    Function: INFA_EXPR_getPrototypeEcho

Description: Gets the arguments of the ECHO function in the Expression
Editor. It calls destroyString to delete the arguments from memory when
usage is complete. The return value must be a null-terminated string. The
function returns NULL if there is no value for the arguments.

Input:  Namespace and name of the function.
Output: Prototype of the function
Remarks: Returns the prototype of function.  The Custom Functions API calls
destroy string to free the allocated memory.
 *************************************************************************/

extern "C" SAMPLE_EXPR_SPEC
    IUNICHAR * getPrototypeEcho(IUNICHAR* ns, IUNICHAR* sFuncName)
{
    static IUNICHAR *uniProt = NULL;
    const char *prototype = "Echo(x), where x can be any type, returns x";

    if (uniProt)
        return uniProt;

    int i, len;
    len = strlen(prototype);
    uniProt = new IUNICHAR[2*len+2];
    for (i=0; i<len; i++)
    {
        uniProt[i] = prototype[i];
    }
    uniProt[i] = 0;
    return uniProt;
}


/*************************************************************************
    Function: validateFunctionEcho

Description: Validates the arguments in the ECHO function. Provides the
```

```
name, datatype, precision, and scale of the arguments in the ECHO function.
Provides the datatype of the return value of the ECHO function. PowerCenter
calls this function once for each instance of the ECHO function used in a
mapping or workflow. Returns ISUCCESS when function usage is valid as per
the syntax.

The ECHO function takes exactly one argument of any datatype. The return
datatype is the same as the input datatype, because the function echoes the
input. Otherwise, returns IFAILURE.

Input: Namespace and name of the function, the number of arguments being
passed, and the metadata (datatype, scale, precision) of each argument.
Output: retValue. Set the metadata for return type.
Remarks: Called by the Custom Functions API to validate the usage of the
function and the input argument metadata to be passed. The plug-in needs
to verify the number of arguments for this function, the expected metadata
for each argument, etc. The plug-in can optionally change the expected datatype
of the input arguments. The plug-in needs to set the return type metadata.
The plugin can specify if the return value of this function is constant,
depending on whether or not all input arguments are constant.
 *****************************************************************************/

extern "C" SAMPLE_EXPR_SPEC
    INFA_EXPR_STATUS validateFunctionEcho(IUNICHAR* ns, IUNICHAR* sFuncName,
                                          IUINT32 numArgs,
                                          INFA_EXPR_OPD_METADATA** inputArgList,
                                          INFA_EXPR_OPD_METADATA* retValue)
{
    INFA_EXPR_STATUS exprStatus;

    // Check number of arguments.
    if (numArgs != 1)
    {
        static const char *err = "Echo function takes one argument.";
        IUNICHAR *errMsg = NULL;

        unsigned int len = strlen(err);
        errMsg = new IUNICHAR[2*len+2];
        unsigned int i;
        for (i=0; i<len; i++)
        {
            errMsg[i] = err[i];
        }
        errMsg[i] = 0;

        exprStatus.status = IFAILURE;
        exprStatus.errMsg = errMsg;
        return exprStatus;
    }


    // This is an echo function.
    // It returns the input value.
    retValue->datatype = inputArgList[0]->datatype;
    retValue->precision = inputArgList[0]->precision;
    retValue->scale = inputArgList[0]->scale;

    // If the input value is constant,
    // the return value is also constant.
    if (inputArgList[0]->isValueConstant)
        retValue->isValueConstant = ITRUE;
    else
        retValue->isValueConstant = IFALSE;


    exprStatus.status = ISUCCESS;
    return exprStatus;
}

/*****************************************************************************
    Function: processRowEcho

Description: Called when an input row is available to an ECHO function
instance. The data for the input arguments of the ECHO function is bound and
accessed through fnInstance-inputOPDHandles. Set the data, length, and
indicator for the output and return ports in fnInstance->retHandle.
```

```
PowerCenter calls the function-level initialization function before calling
this function.

Returns INFA_ROWSUCCESS when the function successfully processes the row of
data. Returns INFA_ROWERROR when the function encounters an error for the row
of data. The Integration Service increments the internal error count.
Only returns this value when the data access mode is row. Returns
INFA_FATALERROR when the function encounters a fatal error for the row of
data or the block of data. The Integration Service fails the session.

Input:  Function instance handle, which has the input data.
Output: return value
Remarks: The plug-in needs to get various input arguments from the function
instance handle, perform calculations, and set the return value.
 *****************************************************************************/

extern "C" SAMPLE_EXPR_SPEC
    INFA_EXPR_ROWSTATUS processRowEcho(INFA_EXPR_FUNCTION_INSTANCE_HANDLE *fnInstance, IUNICHAR
**errMsg)
{

    INFA_EXPR_OPD_RUNTIME_HANDLE* arg1 = fnInstance->inputOPDHandles[0];
    INFA_EXPR_OPD_RUNTIME_HANDLE* retHandle = fnInstance->retHandle;

    // Check if the input argument has a null indicator.
    // If yes, the return value is also null.
    if (INFA_EXPR_GetIndicator(arg1) == INFA_EXPR_NULL_DATA)
    {
        INFA_EXPR_SetIndicator(retHandle, INFA_EXPR_NULL_DATA);
        return INFA_EXPR_SUCCESS;
    }

    short sval;
    long lval;
    int ival;
    char *strval;
    IUNICHAR *ustrval;
    void *rawval;
    float fval;
    double dval;
    INFA_EXPR_DATE *infaDate = NULL;
    int len;

    // Depending on the datatype,
    // get the input argument
    // and set the same value in the return value.
    // Also, set the same indicator.
    switch (arg1->pOPDMetadata->datatype)
    {
        case eCTYPE_SHORT:
            sval = INFA_EXPR_GetShort(arg1);
            INFA_EXPR_SetShort(retHandle, sval);
            INFA_EXPR_SetIndicator(retHandle, INFA_EXPR_GetIndicator(arg1));
            break;

        case eCTYPE_LONG:
        case eCTYPE_LONG64:
            lval = INFA_EXPR_GetLong(arg1);
            INFA_EXPR_SetLong(retHandle, lval);
            INFA_EXPR_SetIndicator(retHandle, INFA_EXPR_GetIndicator(arg1));
            break;

        case eCTYPE_INT32:
            ival = INFA_EXPR_GetInt(arg1);
            INFA_EXPR_SetInt(retHandle, ival);
            INFA_EXPR_SetIndicator(retHandle, INFA_EXPR_GetIndicator(arg1));
            break;

        case eCTYPE_CHAR:
            strval = INFA_EXPR_GetString(arg1);
            len = INFA_EXPR_GetLength(arg1);
            strcpy((char *)retHandle->pUserDefinedPtr, strval);
            INFA_EXPR_SetString(retHandle, retHandle->pUserDefinedPtr);
            INFA_EXPR_SetLength(retHandle, INFA_EXPR_GetLength(arg1));
            INFA_EXPR_SetIndicator(retHandle, INFA_EXPR_GetIndicator(arg1));
            break;
```

```
        case eCTYPE_RAW:
            rawval = INFA_EXPR_GetRaw(arg1);
            len = INFA_EXPR_GetLength(arg1);
            memcpy(retHandle->pUserDefinedPtr, rawval, len);
            INFA_EXPR_SetRaw(retHandle, retHandle->pUserDefinedPtr);
            INFA_EXPR_SetLength(retHandle, len);
            INFA_EXPR_SetIndicator(retHandle, INFA_EXPR_GetIndicator(arg1));
            break;

        case eCTYPE_UNICHAR:
            ustrval = INFA_EXPR_GetUniString(arg1);
            len = INFA_EXPR_GetLength(arg1);
            memcpy(retHandle->pUserDefinedPtr, ustrval, 2*(len+1));
            INFA_EXPR_SetUniString(retHandle, retHandle->pUserDefinedPtr);
            INFA_EXPR_SetLength(retHandle, len);
            INFA_EXPR_SetIndicator(retHandle, INFA_EXPR_GetIndicator(arg1));
            break;

        case eCTYPE_TIME:
            infaDate = INFA_EXPR_GetDate(arg1);
            *((INFA_EXPR_DATE *)retHandle->pUserDefinedPtr) = *infaDate;
            INFA_EXPR_SetDate(retHandle, retHandle->pUserDefinedPtr);
            INFA_EXPR_SetIndicator(retHandle, INFA_EXPR_GetIndicator(arg1));
            break;

        case eCTYPE_FLOAT:
            fval = INFA_EXPR_GetFloat(arg1);
            INFA_EXPR_SetFloat(retHandle, fval);
            INFA_EXPR_SetIndicator(retHandle, INFA_EXPR_GetIndicator(arg1));
            break;

        case eCTYPE_DOUBLE:
            dval = INFA_EXPR_GetDouble(arg1);
            INFA_EXPR_SetDouble(retHandle, dval);
            INFA_EXPR_SetIndicator(retHandle, INFA_EXPR_GetIndicator(arg1));
            break;

        default:
            return INFA_EXPR_ROWERROR;
            break;
    }
    return INFA_EXPR_SUCCESS;
}

/*****************************************************************************
    Function: moduleInitEcho

Description: Called once for each module to initialize any global data
structure in the function. Called before calling any function-level
functions. Returns ISUCCESS when module initialization is successful.
Otherwise, returns IFAILURE.

Input:  module handle
Output: status
Remarks: The plug-in can optionally implement this method for one-time
initialization.
 *****************************************************************************/

extern "C" SAMPLE_EXPR_SPEC
    INFA_EXPR_STATUS moduleInitEcho(INFA_EXPR_MODULE_HANDLE *modHandle)
{
    INFA_EXPR_STATUS exprStatus;

    // initialize the ECHO_STR
    const char *fnName = "Echo";
    int len = strlen(fnName);
    int i;
    for (i=0;i<len;i++)
        ECHO_STR[i] = fnName[i];

    exprStatus.status = ISUCCESS;
    return exprStatus;
}

/*****************************************************************************
```

```
    Function: moduleDeinitEcho

Description: Called once for each module to deinitialize any data structure
in this function. Called after all function-level interactions are complete.
Returns ISUCCESS when module deinitialization is successful. Otherwise,
returns IFAILURE.

Input:  module handle
Output: status
Remarks: The plug-in can optionally implement this method for one-time
deinitialization.
 ****************************************************************************/

extern "C" SAMPLE_EXPR_SPEC
    INFA_EXPR_STATUS moduleDeinitEcho(INFA_EXPR_MODULE_HANDLE *modHandle)
{
    INFA_EXPR_STATUS exprStatus;
    exprStatus.status = ISUCCESS;
    return exprStatus;

}

/****************************************************************************
    Function: functionInitEcho

Description: Called once for each custom function to initialize any
structure related to the custom function. Module-level initialization
function is called before this function. Returns ISUCCESS when function
init is successful. Otherwise, returns IFAILURE.

Input:  function handle
Output: status
Remarks: The plug-in can optionally implement this method for one-time function
initialization.
 ****************************************************************************/

extern "C" SAMPLE_EXPR_SPEC
    INFA_EXPR_STATUS functionInitEcho(INFA_EXPR_FUNCTION_HANDLE *funHandle)
{
    INFA_EXPR_STATUS exprStatus;
    exprStatus.status = ISUCCESS;
    return exprStatus;

}

/****************************************************************************
    Function: functionDeinitEcho

Description: Called once for each function level to deinitialize any
structure related to the ECHO function. Returns ISUCCESS when function deinit
is successful. Otherwise, returns IFAILURE.

Input:  function handle
Output: status
Remarks: The plug-in can optionally implement this method for one-time function
deinitialization.
 ****************************************************************************/

extern "C" SAMPLE_EXPR_SPEC
    INFA_EXPR_STATUS functionDeinitEcho(INFA_EXPR_FUNCTION_HANDLE *funHandle)
{
    INFA_EXPR_STATUS exprStatus;
    exprStatus.status = ISUCCESS;
    return exprStatus;

}

/****************************************************************************
    Function: functionInstInitEcho

Description: Called once for each custom function instance to initialize
any structure related to the an instance of the ECHO function. If there are
two instances of ECHO in a mapping or workflow, PowerCenter calls this
function twice. PowerCenter calls the module-level initialization function
before calling this function.  Returns ISUCCESS when function instance
initialization is successful. Otherwise, returns IFAILURE.
```

```
Input:  function instance handle
Output: status
Remarks: The plug-in can optionally implement this method for one-time
function instance initialization.
 ********************************************************************************/

extern "C" SAMPLE_EXPR_SPEC
    INFA_EXPR_STATUS functionInstInitEcho(INFA_EXPR_FUNCTION_INSTANCE_HANDLE *funInstHandle)
{
    INFA_EXPR_STATUS exprStatus;
    exprStatus.status = ISUCCESS;

    INFA_EXPR_OPD_RUNTIME_HANDLE *retHandle = funInstHandle->retHandle;

    // Allocate memory depending on the datatype.
    if (retHandle->pOPDMetadata->datatype == eCTYPE_CHAR)
        retHandle->pUserDefinedPtr = new char[retHandle->pOPDMetadata->precision+1];
    else if (retHandle->pOPDMetadata->datatype == eCTYPE_UNICHAR)
        retHandle->pUserDefinedPtr = new IUNICHAR[retHandle->pOPDMetadata->precision+1];
    else if (retHandle->pOPDMetadata->datatype == eCTYPE_RAW)
        retHandle->pUserDefinedPtr = new unsigned char[retHandle->pOPDMetadata->precision];
    else if (retHandle->pOPDMetadata->datatype == eCTYPE_TIME)
        retHandle->pUserDefinedPtr = new INFA_EXPR_DATE();
    return exprStatus;
}

/********************************************************************************
    Function: functionInstDeinitEcho

Description: Called once for each function level during deinitialization.
Can deinitialize any structure related to the ECHO function. Returns ISUCCESS
when deinitialization is successful. Otherwise, returns IFAILURE.

Input:  function instance handle
Output: status
Remarks: The plug-in can optionally implement this method for one-time
function instance deinitialization.
 ********************************************************************************/

extern "C" SAMPLE_EXPR_SPEC INFA_EXPR_STATUS functionInstDeinitEcho(INFA_EXPR_FUNCTION_INSTANCE_HANDLE
*funInstHandle)
{
    INFA_EXPR_STATUS exprStatus;
    exprStatus.status = ISUCCESS;
    INFA_EXPR_OPD_RUNTIME_HANDLE *retHandle = funInstHandle->retHandle;

    if (retHandle->pOPDMetadata->datatype == eCTYPE_CHAR)
        delete [] (char *)retHandle->pUserDefinedPtr;
    else if (retHandle->pOPDMetadata->datatype == eCTYPE_UNICHAR)
        delete [] (IUNICHAR *)retHandle->pUserDefinedPtr;
    else if (retHandle->pOPDMetadata->datatype == eCTYPE_RAW)
        delete [] (unsigned char *)retHandle->pUserDefinedPtr;
    else if (retHandle->pOPDMetadata->datatype == eCTYPE_TIME)
        delete (INFA_EXPR_DATE *)retHandle->pUserDefinedPtr;
    return exprStatus;
```

Step 4. Build the Modules

You can build the modules on Windows or UNIX. Build a module for each platform that PowerCenter runs on. You must build a module on Windows, because the PowerCenter Client resides on Windows. You may also need to build modules on UNIX or Linux, depending on the node that hosts the Integration Service.

The following table lists the library file names for each platform when you build the module:

Platform	Module File Name
Windows	<module_identifier>.dll
AIX	lib<module_identifier>.a
HP-UX	lib<module_identifier>.sl
Linux	lib<module_identifier>.so
Solaris	lib<module_identifier>.so

You declare these modules in the repository plug-in XML file.

Related Topics:

- "Step 5. Create the Repository Plug-in File" on page 189

Building the Module on Windows

On Windows, you can use Microsoft Visual C++ to build the module.

To build the module on Windows:

1. Start Visual C++.
2. Click File > New.
3. In the New dialog box, click the Projects tab and select the Win32 Dynamic-Link Library option.
4. Enter its location.

   In the Echo example, enter server\samples\exprsdk\echo.
5. Enter the name of the project.

   You must use the module name specified for the custom function as the project name. In the Echo example, enter EchoDemo.
6. Click OK.

   Visual C++ creates a wizard to define the project components.
7. In the wizard, select an empty DLL project and click Finish. Click OK in the New Project Information dialog box.

   Visual C++ creates the project files in the directory you specified.
8. Click Project > Add To Project > Files.
9. Navigate up a directory level. This directory contains the procedure files you created. Select all .c files and click OK.

   In the Echo example, add the Echo.c file.
10. Click Project > Settings.
11. Click the C/C++ tab, and select Preprocessor from the Category field.
12. In the Additional Include Directories field, enter the following path and click OK:

    ```
    ..; <Integration Service_installation_directory>\download\C function Samples\Echo...
    ```
13. Click Build > Build <module_name>.dll or press F7 to build the project.

    Visual C++ creates the DLL and places it in the debug or release directory under the project directory.

Building the Module on UNIX

On UNIX, you can use any C compiler to build the module.

To build the module on UNIX:

1. Set the environment variable INFA_HOME to the Integration Service installation directory.

   Note: If you specify an incorrect directory path for the INFA_HOME environment variable, the Integration Service cannot start.

   Restart the node to apply changes.

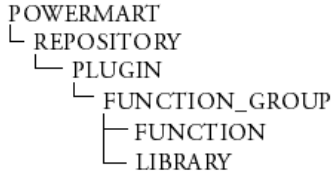
2. Enter a command from the following table to make the project.

UNIX Version	Command
AIX (32-bit)	make -f makefile.aix
AIX (64-bit)	make -f makefile.aix64
HP-UX (32-bit)	make -f makefile.hp
HP-UX (64-bit)	make -f makefile.hp64
HP-UX PA-RISC	make -f makefile.hpparisc64
Linux	make -f makefile.linux
Solaris	make -f makefile.sol

Step 5. Create the Repository Plug-in File

Create an XML file to define the function metadata of one more custom functions. Use the structure of the plug-in DTD file when you create or modify the plug-in XML file. The plug-in DTD file, plugin.dtd, is stored in the PowerCenter Client directory. Use any tool that can create an XML file. When you create the repository plug-in file, provide a new name for the file.

The following figure shows the structure of plugin.dtd:



The PLUGIN Element

Use the PLUGIN element to define the metadata for the plug-in that you want to create. The attributes of the PLUGIN element uniquely identify the plug-in metadata.

The following table shows the attributes of the PLUGIN element:

Attribute	Required/ Optional	Description
NAME	Required	Name of the plug-in. The plug-in name displays on the Plugin tab of the Repository Service.
ID	Required	ID of the plug-in. Use to distinguish plug-ins developed using the same VENDORID.
VENDORNAME	Required	Name of the vendor. The vendor name displays on the Plugin tab of the Repository Service.
VENDORID	Required	Vendor ID. Get a vendor ID from Informatica if you are developing custom functions to distribute outside your organization. For more information, see “Step 1. Get Repository ID Attributes” on page 177.
DESCRIPTION	Optional	Description of the plug-in. The plug-in description displays on the Plugin tab of the Repository Service.
VERSION	Required	Version of the plug-in. Use to track updates to the plug-in metadata.

The FUNCTION_GROUP Element

Use the FUNCTION_GROUP element to define the group the custom function belongs to.

The following table shows the attributes of the FUNCTION_GROUP element:

Attribute	Required/ Optional	Description
NAME	Required	Name of the custom function group that you want to define. The function group name displays on the Plugin tab of the Repository Service.
ID	Required	ID for the function group. Get a function group ID from Informatica if you are developing custom functions to distribute outside your organization. For more information, see “Step 1. Get Repository ID Attributes” on page 177. The function group ID displays on the Plugin tab of the Repository Service.
COMPONENTVERSION	Required	Version number of the function group. This tracks updates to the FUNCTION_GROUP element.
DESCRIPTION	Optional	Description of the function group. The function group description displays on the Plugin tab of the Repository Service.
NAMESPACE	Required	Namespace of the function group. The Expression Editor displays custom functions with the namespace in a separate folder on the Functions tab. Namespaces are not case sensitive. You cannot use the namespace “infa.” It is reserved. Also, the namespace cannot be empty.

Determining a Namespace

You can choose one namespace for all functions you create. However, the namespace cannot conflict with the namespace of custom functions developed by other vendors. Therefore, choose a unique namespace. For example, you can select a namespace that pertains to your company name, such as its stock symbol.

The FUNCTION Element

Use the FUNCTION element to define properties of the custom function.

The following table shows the attributes of the FUNCTION element:

Attribute	Required/ Optional	Description
NAME	Required	Name of the third-party function that you want to define.
ID	Required	ID for FUNCTION element. Identifies the function. Get a function ID from Informatica if you are developing custom functions to distribute outside your organization. For more information, see “Step 1. Get Repository ID Attributes” on page 177.
FUNCTION_CATEGORY	Optional	Category of the function you want to define. Use one of the following categories: - Character - Conversion - Data Cleaning - Date - Numerical - Scientific - Special - Test The Expression Editor displays the custom function under this category.

The LIBRARY Element

Use the LIBRARY element to specify the compiled shared libraries for the custom function.

The following table shows the attributes of the LIBRARY element:

Attribute	Required/ Optional	Description
NAME	Required	Name of the compiled shared library.
OSTYPE	Required	Operating system for which you compiled the shared library.
TYPE	Required	Type of shared library. Specify one of the following types: - VALIDATION. Library the PowerCenter Client uses to retrieve the custom function description and validate the function invocation, such as the return type and number of arguments. - SERVER. Library the Integration Service uses to execute the function call.

Sample Plug-in XML File

The following example shows the repository plug-in file that defines the ECHO custom function:

```
<?xml version="1.0" encoding="us-ascii"?>
<!DOCTYPE POWERMART SYSTEM "plugin.dtd">
<POWERMART>
    <REPOSITORY CODEPAGE="us-ascii">
        <PLUGIN NAME="Echo" ID="506001" VENDORNAME="Informatica"
            VENDORID="1"
            DESCRIPTION="Plugin for Expressions from Informatica">
            <FUNCTION_GROUP ID="506002" NAME="INFA Function Group1"
                COMPONENTVERSION="1.0.0"
                DESCRIPTION="The functions group for my own Echo function"
                NAMESPACE="">
               <FUNCTION ID="506004" NAME="ECHO" FUNCTION_CATEGORY="Data Cleansing"/>
               <LIBRARY NAME="pmecho.dll" OSTYPE="NT" TYPE="VALIDATION"/>
               <LIBRARY NAME="llibpmecho.sl" OSTYPE="HPUX" TYPE="VALIDATION"/>
               <LIBRARY NAME="libpmecho.so" OSTYPE="SOLARIS" TYPE="VALIDATION"/>
               <LIBRARY NAME="libpmecho.so " OSTYPE="LINUX" TYPE="VALIDATION"/>
               <LIBRARY NAME="libpmecho.a" OSTYPE="AIX" TYPE="VALIDATION"/>
               <LIBRARY NAME="pmecho.dll" OSTYPE="NT" TYPE="SERVER"/>
               <LIBRARY NAME="libpmecho.sl" OSTYPE="HPUX" TYPE="SERVER"/>
               <LIBRARY NAME="libpmecho.so" OSTYPE="SOLARIS" TYPE="SERVER"/>
               <LIBRARY NAME="libpmecho.so" OSTYPE="LINUX" TYPE="SERVER"/>
               <LIBRARY NAME="libpmecho.a" OSTYPE="AIX" TYPE="SERVER"/>
</FUNCTION_GROUP>
        </PLUGIN>
    </REPOSITORY>
</POWERMART>
```

Step 6. Test Custom Functions

You can test custom functions during development. Complete the following tasks to test custom functions in PowerCenter:

- Validate the repository plug-in XML file.
- Verify that custom functions in an expression produce accurate data.

To test custom functions, you must install the custom functions in a PowerCenter environment.

Validating the Repository Plug-in File

You can validate the repository plug-in file by registering it in a PowerCenter repository. When you register a plug-in file, an associated DTD file called plugin.dtd validates the structure of the file. The file must conform to the structure of the associated plugin.dtd. plugin.dtd is in the PowerCenter Client directory.

When you develop custom functions, you can create a repository plug-in file and register it before you finish creating the header and implementation files for the functions. When you register the file, you add custom function metadata, such as the plug-in ID, namespace, and function names. This reserves this information in the repository.

After you register the repository plug-in file, you can continue to develop custom functions. When you finish developing the functions, register the repository plug-in file again to update the custom function metadata in the repository.

After you register the repository plug-in, you can view the plug-in metadata.

Viewing Plug-in Metadata Details

The following table shows the metadata that the Informatica Administrator displays on the Plug-ins tab:

Repository Service Attribute	XML Element and Attribute
Name	PLUGIN NAME
Vendor name	PLUGIN VENDORNAME
Description	PLUGIN DESCRIPTION
Group name	FUNCTION_GROUP NAME
Group ID	FUNCTION_GROUP ID
Group Description	FUNCTION_GROUP DESCRIPTION

Verifying Function Accuracy

To verify the accuracy of a custom function, create an expression with the function and include it in a mapping and workflow. Complete the following steps to verify the accuracy of a custom function:

1. Create test data.
2. Create a mapping.
3. Add the custom function to an expression in the mapping.
4. Create a mapping.
5. Run the Debugger (optional). Or, create a session and workflow for the mapping.
6. Run the workflow.
7. View the results.

Installing Custom Functions

Complete the following steps to install custom functions:

1. Copy the custom function libraries to the PowerCenter environment.
2. Register the repository plug-in.

Once you install custom functions, use them in transformation and workflow expressions.

Step 1. Copy Custom Function Libraries to PowerCenter

Copy the custom function libraries and the repository plug-in XML file to the PowerCenter Client and Integration Service directories per the custom function developer instructions.

If you have high availability or run sessions on a grid, put the libraries in a single location and define the location as a required resource.

Step 2. Register the Plug-in

Register the repository plug-in XML file from the Administrator tool.

Creating Expressions with Custom Functions

You can add a custom function to an expression. If you enter a custom function when you manually create an expression, you must prefix the user-defined function with the namespace the custom function developer provides. When you create an expression with the Expression Editor, custom functions display in the list of all functions and with their function type. Use custom functions as you would any other function.

When you validate the expression, the Designer or Workflow Manager does not validate the custom function. They only validate the expression. The plug-in validates the custom function.

CHAPTER 8

Custom Function API Reference

This chapter includes the following topics:

Custom Function API Reference Overview

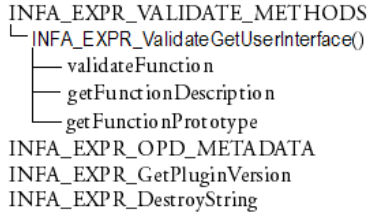
Use the Custom Function API to develop custom functions that you can include in a transformation or workflow expression. The Custom Function API is a framework to create custom functions. It includes common and run-time APIs. The APIs enable PowerCenter to validate expressions with custom functions and use the expressions in workflows.

Use the APIs in the header and implementation files to develop custom functions. You must build shared libraries with the header and implementation files. You specify the shared libraries in a repository plug-in file that you register in PowerCenter. You also copy the shared libraries to the PowerCenter environment.

Common APIs

The PowerCenter Client, Integration Service, and Repository Service call the common APIs to validate expressions, delete function returns from memory after use, and delete function descriptions and prototypes from memory after use.

The common APIs contain the following structure:



Validation Handle

The INFA_EXPR_VALIDATE_METHODS handle is a validation handle. PowerCenter calls INFA_EXPR_ValidateGetUserInterface to get function pointers in this validation handle.

User Interface Validation Function

When PowerCenter calls INFA_EXPR_ValidateGetUserInterface, the plug-in returns function pointers to the validation functions.

Use the following syntax:

```
INFA_EXPR_STATUS INFA_EXPR_ValidateGetUserInterface( IUNICHAR* sNamespace, IUNICHAR* sFuncName,
INFA_EXPR_VALIDATE_METHODS* functions);
```

Argument	Datatype	Input/ Output	Description
sNamespace	IUNICHAR	Input	Namespace of the custom function.
sFuncName	IUNICHAR	Input	Name of the custom function.
functions	INFA_EXPR_VALIDATE_METHODS	Output	Pointers to different functions called during validation and reporting.

The return datatype is `INFA_EXPR_STATUS`. Use ISUCCESS and IFAILURE as the return value. When the function returns IFAILURE, the plug-in did not implement the function or another error occurred.

INFA_EXPR_ValidateGetUserInterface returns the following functions:

- **validateFunction.** Validates a custom function.
- **getFunctionDescription.** Describes a custom function.
- **getFunctionPrototype.** Provides the prototype for a custom function.

Custom Function Validation Function

PowerCenter calls validateFunction to validate the arguments in the custom function. It uses this function to provide the name, datatype, precision, and scale of the arguments in the custom function. It also uses this function to provide the datatype of the return value of the custom function.

PowerCenter calls this function once for each instance of the custom function used in a mapping or workflow.

Use the following syntax:

```
INFA_EXPR_STATUS *(validateFunction)(IUNICHAR* sNamespace, IUNICHAR* sFuncName, IUINT32 numArgs,
INFA_EXPR_OPD_METADATA** inputArgList, INFA_EXPR_OPD_METADATA* retValue);
```

Argument	Datatype	Input/ Output	Description
sNamespace	IUNICHAR	Input	Namespace of the function.
sFuncName	IUNICHAR	Input	Name of the custom function to validate.
numArgs	IUINT32	Input	Number of arguments in the custom function.

Argument	Datatype	Input/ Output	Description
inputArgList	INFA_EXPR_OPD_METADATA	Input	Input arguments of the custom function.
retValue	INFA_EXPR_OPD_METADATA	Output	Metadata of the return port of the custom function. Set the datatype, precision, and scale of the return value in this argument.

The return datatype is `INFA_EXPR_STATUS`. Use ISUCCESS and IFAILURE as the return value. When the function returns IFAILURE, PowerCenter displays an error message.

Custom Function Description Function

PowerCenter calls getFunctionDescription to get a description of the custom function. It calls destroyString to delete the description from memory when usage is complete.

Use the following syntax:

```
IUNICHAR* *(getFunctionDescription) (IUNICHAR* sNamespace, IUNICHAR* sFuncName);
```

Argument	Datatype	Input/ Output	Description
sNamespace	IUNICHAR	Input	Namespace of the function.
sFuncName	IUNICHAR	Input	Name of the custom function the plug-in should describe.

The return datatype is IUNICHAR. The return value must be a null-terminated string.

Custom Function Prototype Function

PowerCenter calls getFunctionPrototype to get the arguments of the custom function in the Expression Editor. It calls destroyString to delete the arguments from memory when usage is complete.

Use the following syntax:

```
IUNICHAR* *(getFunctionPrototype) (INICHAR* sNamespace, IUNICHAR* sFuncName);
```

Argument	Datatype	Input/ Output	Description
sNamespace	IUNICHAR	Input	Namespace of the function.
sFuncName	IUNICHAR	Input	Name of the custom function the plug-in should describe.

The return datatype is IUNICHAR. The return value must be a null-terminated string. The function returns NULL if there is no value for the arguments.

INFA_EXPR_OPD_METADATA Structure

This structure defines the metadata of the expression operands, including arguments passed to the function and the return type.

The structure contains the following metadata:

- **datatype.** Datatype of the argument.
- **precision.** Precision of the argument.
- **scale.** Scale of the argument.
- **isValueConstant.** Indicates if the argument is a constant. If so, the framework evaluates the argument once for each function call. The framework uses isValueConstant to optimize for performance. For input arguments that are constants, the plug-in can get the argument values during function instance initialization to optimize performance. For output values, the plug-in sets isValueConstant to TRUE.

Get Plug-in Version Function

This function defines the version of the plug-in. It must be the same as the Custom Function API version, which is 1.0.0.

Use the following syntax:

```
INFA_EXPR_STATUS INFA_EXPR_GetPluginVersion(INFA_VERSION *sdkVersion, INFA_VERSION *pluginVersion);
```

Argument	Datatype	Input/ Output	Description
sdkVersion	INFA_VERSION	Input	Version of the Custom Function API. Use 1.0.0.
pluginVersion	INFA_VERSION	Output	Version of the plug-in you want to create.

The return datatype is INFA_EXPR_STATUS. Use ISUCCESS and IFAILURE as the return values. If the function returns IFAILURE, the session or workflow fails.

Destroy String Function

This function destroys all strings the plug-in returns. For example, it destroys error messages or the return value of other function calls, such as getFunctionDescription.

Use the following syntax:

```
void *(DestroyString)(IUNICHAR* str);
```

Argument	Datatype	Input/ Output	Description
str	IUNICHAR	Input	The input string this function deletes.

The function returns no value.

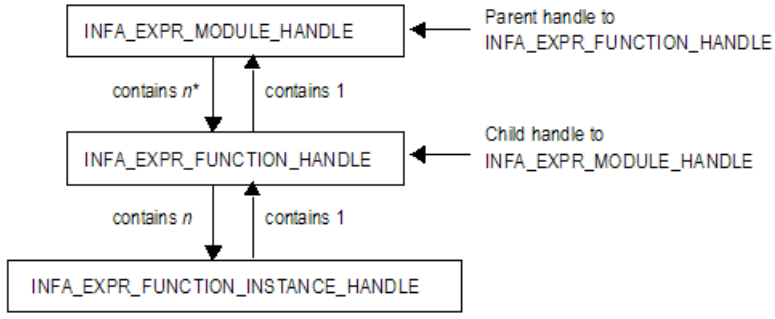
Run-time APIs

The Integration Service calls the run-time APIs during a session to evaluate the expression that contains the custom function. It initializes the plug-in at the module, function, and function instance levels.

Each level contains a set of functions. These functions are associated with a handle, such as INFA_EXPR_MODULE_HANDLE. The first parameter for these functions is the handle the function affects.

Custom Function API handles have a hierarchical relationship to each other. A parent handle has a 1:*n* relationship to its child handle.

The following figure shows the Custom Function API handles:



The following table describes the run-time handles:

Handle Name	Description
INFA_EXPR_MODULE_HANDLE	Represents the shared library or DLL. The plug-in can only access the module handle in its own shared library or DLL. It cannot access the module handle in any other shared library or DLL.
INFA_EXPR_FUNCTION_HANDLE	Represents a custom function within the shared library or DLL.
INFA_EXPR_FUNCTION_INSTANCE_HANDLE	Represents a specific custom function instance.

Module-Level Functions

PowerCenter calls module-level functions once for each shared library or DLL.

Get User Interface Module-Level Function

This function sets the function pointers for module-level interaction.

Use the following syntax:

```
INFA_EXPR_STATUS INFA_EXPR_ModuleGetUserInterface(INFA_EXPR_LIB_METHODS* functions);
```

Argument	Datatype	Input/ Output	Description
functions	INFA_EXPR_LIB_METHODS	Output	Module get user interface functions.

The return datatype is INFA_EXPR_STATUS. Use ISUCCESS and IFAILURE as the return values. If the function returns IFAILURE, the session or workflow fails.

This function returns the following functions:

- **function_init.** Initializes the function.
- **function_deinit.** Deinitializes the function.

Module-Level Initialization Function

PowerCenter calls this module_init once for each module to initialize any global data structure in the function. It calls this function before calling any function-level functions.

Use the following syntax:

```
INFA_EXPR_STATUS (*module_init) (INFA_EXPR_MODULE_HANDLE module);
```

Argument	Datatype	Input/ Output	Description
module	INFA_EXPR_MODULE_HANDLE	Input	Stores data that it can retrieve at the function level.

The return datatype is INFA_EXPR_STATUS. Use ISUCCESS and IFAILURE as the return values. If the function returns IFAILURE, the session or workflow fails.

Module-Level Deinitialization Function

PowerCenter calls module_deinit once for each module to deinitialize any data structure in this function. It calls this function after all function-level interactions are complete.

Use the following syntax:

```
INFA_EXPR_STATUS (*module_deinit) (INFA_EXPR_MODULE_HANDLE module);
```

Argument	Datatype	Input/ Output	Description
module	INFA_EXPR_MODULE_HANDLE	Input	Module-level handle that the framework passes to the plug-in when the module init function is called.

The return datatype is INFA_EXPR_STATUS. Use ISUCCESS and IFAILURE as the return values. If the function returns IFAILURE, the session or workflow fails.

Function-Level Functions

PowerCenter calls the function-level functions once for each custom function and once for each shared library or DLL that provides the parameters for the custom function.

Get User Interface Function-Level Function

This function sets the function pointers for function-level interaction. PowerCenter calls this function for every custom function this library implements.

Use the following syntax:

```
INFA_EXPR_STATUS INFA_EXPR_FunctionGetUserInterface (IUNICHAR* nameSpaceName, IUNICHAR* functionName,
INFA_EXPR_FUNCTION_METHODS* functions);
```

Argument	Datatype	Input/ Output	Description
nameSpaceName	IUNICHAR	Input	Namespace of the function.
functionName	IUNICHAR	Input	Name of the custom function the plug-in should describe.
function	INFA_EXPR_FUNCTION_METHODS	Input	Place holder for the function pointers to be invoked at the function instance-level.

The return datatype is INFA_EXPR_STATUS. Use ISUCCESS and IFAILURE as the return values. If the function returns IFAILURE, the session or workflow fails.

This function returns the following functions:

- **function_init.** Initializes the function.
- **function_deinit.** Deinitializes the function.

Function-Level Initialization Function

PowerCenter calls function_init once for each custom function to initialize any structure related to the custom function. It calls the module-level initialization function before calling this function.

Use the following syntax:

```
INFA_EXPR_STATUS (*function_init) (INFA_EXPR_FUNCTION_HANDLE fnInstance);
```

Argument	Datatype	Input/ Output	Description
fnInstance	INFA_EXPR_FUNCTION_HANDLE	Input	Performs the following tasks: - Stores user-defined pointers for the framework to retrieve during run time or deinitialization. - Initializes data structures for the function instance-level. - If the input argument is a constant, the plug-in retrieves this constant value and performs any necessary preprocessing.

The return datatype is INFA_EXPR_STATUS. Use ISUCCESS and IFAILURE as the return values. If the function returns IFAILURE, the session or workflow fails.

Function-Level Deinitialization Function

PowerCenter calls this function once for each function level to deinitialize any structure related to the custom function.

Use the following syntax:

```
INFA_EXPR_STATUS (*function_deinit) (INFA_EXPR_FUNCTION_HANDLE function);
```

Argument	Datatype	Input/ Output	Description
fnInstance	INFA_EXPR_FUNCTION_HANDLE	Input	Function-level handle that the framework passes to the plug-ins when the function instance-level init function is called.

The return datatype is INFA_EXPR_STATUS. Use ISUCCESS and IFAILURE as the return values. If the function returns IFAILURE, the session or workflow fails.

Function Instance-Level Functions

PowerCenter calls these functions each time a custom function is used in a mapping or workflow.

Get User Interface Function-Level Function

This function sets the function pointers for function-level interaction. PowerCenter calls this function for every custom function this library implements.

Use the following syntax:

```
INFA_EXPR_STATUS INFA_EXPR_FunctionInstanceGetUserInterface(IUNICHAR* functionName,
INFA_EXPR_FUNCTION_INSTANCE_METHODS* functions)
```

Argument	Datatype	Input/ Output	Description
functionName	IUNICHAR	Input	Namespace of the function.
functions	INFA_EXPR_FUNCTION_INSTANCE_METHODS	Input	Place holder for the function pointers to be invoked at the function instance-level.

This function returns the following functions:

- **fnInstance_init.** Initializes an instance of a custom function.
- **fnInstance_processRow.** Processes data for an instance of the custom function.
- **fnInstance_deinit.** Deinitializes an instance of a custom function.

Function Instance-Level Initialization Function

PowerCenter calls fnInstance_init once for each custom function instance to initialize any structure related to the custom function instance. If there are two instances of a custom function in a mapping or workflow, PowerCenter calls this function twice. PowerCenter calls the module-level initialization function before calling this function.

Use the following syntax:

```
INFA_EXPR_STATUS (*fnInstance_init)(INFA_EXPR_FUNCTION_INSTANCE_HANDLE fnInstance);
```

Argument	Datatype	Input/ Output	Description
fnInstance	INFA_EXPR_FUNCTION_HANDLE	Input	Performs the following tasks: - Stores user-defined pointers for the framework to retrieve during run time or deinitialization. - Initializes data structures for the function instance level. - If the input argument is a constant, the plug-in retrieves this constant value and for performs any necessary preprocessing.

The return datatype is INFA_EXPR_STATUS. Use ISUCCESS and IFAILURE as the return values. If the function returns IFAILURE, the session or workflow fails.

Function Instance Row Processing Function

PowerCenter calls this fnInstance_processRow when an input row is available to a custom function instance. The data for the input arguments of the custom function is bound and accessed through fnInstance-inputOPDHandles. Set the data, length, and indicator for the output and return ports in fnInstance->retHandle. PowerCenter calls the function-level initialization function before calling this function.

Use the following syntax:

```
INFA_EXPR_ROWSTATUS (*fnInstance_processRow) (INFA_EXPR_FUNCTION_INSTANCE_HANDLE fnInstance);
```

Argument	Datatype	Input/ Output	Description
fnInstance	INFA_EXPR_FUNCTION_HANDLE	Input	Function-level handle for which data is available.

The datatype of the return value is INFA_EXPR_ROWSTATUS. Use the following values for the return value:

- **INFA_ROWSUCCESS.** Indicates the function successfully processed the row of data.
- **INFA_ROWERROR.** Indicates the function encountered an error for the row of data. The Integration Service increments the internal error count. Only return this value when the data access mode is row.
- **INFA_FATALERROR.** Indicates the function encountered a fatal error for the row of data or the block of data. The Integration Service fails the session.

Function Instance-Level Deinitialization Function

PowerCenter calls fnInstance_deinit once for each function level during deinitialization. It can call this function to deinitialize any structure related to the custom function.

Use the following syntax:

```
INFA_EXPR_STATUS (*fnInstance_deinit)(INFA_EXPR_FUNCTION_INSTANCE_HANDLE fnInstance);
```

Argument	Datatype	Input/ Output	Description
fnInstance	INFA_EXPR_FUNCTION_INSTANCE_HANDLE	Input	Function-level handle that the framework passes to the plug-ins when the function instance-level initialization function is called.

The return datatype is INFA_EXPR_STATUS. Use ISUCCESS and IFAILURE as the return values. If the function returns IFAILURE, the session or workflow fails.

INDEX

D

E

F

M

N

O

P

Q

R

S

T

U

V

W

Y