IBM WebSphere Transformation Extender

Functions and Expressions

Version 8.1

Note
Before using this information, be sure to read the general information in "Notices" on page 209.

October 2006

This edition of this document applies to IBM WebSphere Transformation Extender Version 8.1; and to all subsequent releases and modifications until otherwise indicated in new editions.

To send us your comments about this document, email DTX_doc_feedback@us.ibm.com. We look forward to hearing from you.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Chapter 1. Expressions and evaluations . . . 1

Expressions . . . 1
- Component rule expressions evaluate to true or false . . . 1
- Map rule expressions evaluate to data . . . 1

Literals . . . 1

Data object names . . . 2
- Object names in map rules . . . 2
- Object names in component rules . . . 2
- Card name . . . 3
- Local type name . . . 3
- Partition list . . . 3
- Component path . . . 4
- Indexed object names . . . 4
- Component paths separated by a colon . . . 4
- Component paths separated by IN . . . 5
- Comment object name . . . 5
- Shorthand notation . . . 6

Evaluating expressions . . . 6
- Card order can influence the order of evaluation sets . . . 7
- Functions influence the number of evaluation sets . . . 8
- Object names influence the number of evaluation sets . . . 9

Operators . . . 10
- Arithmetic operators . . . 11
- Text operators . . . 11
- Logical operators . . . 11
- Comparison operators . . . 11
- Order of operator evaluation . . . 11
- Operands . . . 12

Using functions in expressions . . . 12
- Function arguments . . . 12
- Input arguments . . . 13
- Nested input arguments . . . 13
- Output arguments . . . 14
- Function arguments and evaluation . . . 14

Map names in expressions . . . 15
- Evaluation of functional maps in an expression . . . 15

Expressions that evaluate to NONE . . . 16
- When an operand evaluates to NONE . . . 16
- When an input argument of a function evaluates to NONE . . . 16
- When an input argument of a functional map evaluates to NONE . . . 17
- When an input of an executable map evaluates to NONE . . . 17
- Impact of track setting on order of output . . . 17

Evaluated expressions assigned to output items . . . 18
- NONE assigned to an output number . . . 18

Automatic item format conversions . . . 18
- Numeric precision . . . 19

Chapter 2. Using functions . . . 21

Functions in a component rule . . . 21

Functions in a map rule . . . 21

Chapter 3. Syntax of a function . . . 23

Chapter 4. Function argument syntax . . . 25

Chapter 5. General functions 27
CLONE. 27
DEFAULT 27
ECHOIN 28
HANDLEIN 28
REFORMAT 30

Chapter 6. Bit manipulation and testing functions 31
SETOFF 31
SETON 31
TESTOFF 32
TESTON 33

Chapter 7. Conversion functions 35
BASE64TOTEXT 35
BCDTOHEX 35
BCDTOINT 36
BCDTOTEXT 37
CONVERT. 38
DATETONUMBER 38
DATETOTEXT 39
FROMBASETEN. 40
FROMDATETIME 41
FROMNUMBER 41
HEXTEXTTOSTREAM 42
INT 43
NUMBERTODATE 44
NUMBERTOTEXT 44
PACK 45
PACKAGE. 46
QUOTEDTOTEXT 46
SERIESTOTEXT 47
STREAMTOHEXTEXT 48
SYMBOL 49
TEXTTOBASE64 50
TEXTTOBCD 50
TEXTTODATE 51
TEXTTONUMBER 52
TEXTTOQUOTED 52
TEXTTOTIME 53
TEXTTOTIME 54
TOBASETEN 54
TODATETIME 55
TONUMBER 56
UNPACK 57
UNZONE 58
ZONE 59

Chapter 8. Date/time functions 63
ADDDAYS 63
ADDHOURS 64
ADDMINUTES 64
CURRENTDATE. 65
CURRENTDATETIME 66
CURRENTTIME 67
DATETONUMBER 67
DATETOTEXT 68
FROMDATETIME 69
MAX 70
MIN 70

NUMBERTODATE . . . 71
TEXTTODATE . . . 72
TEXTTOTIME . . . 72
TIMETOTEXT . . . 73
TODATETIME . . . 74

Chapter 9. Error handling functions . . . 75
CONTAINSERRORS . . . 75
FAIL. . . . 75
ISERROR . . . 76
ONERROR . . . 77
REJECT. . . . 78
VALID . . . 79

Chapter 10. External interface functions . . . 81
DBLOOKUP . . . 81
DBQUERY. . . . 83
DDEQUERY . . . 86
EXIT. . . . 87
GET . . . 89
PUT . . . 90
RUN . . . 91

Chapter 11. Inspection functions . . . 95
ABSENT . . . 95
CONTAINSERRORS . . . 95
ISALPHA . . . 96
ISERROR . . . 97
ISLOWER . . . 97
ISNUMBER . . . 98
ISUPPER . . . 99
MEMBER . . . 99
NOT . . . 100
OFFSET . . . 101
OR . . . 101
PARTITION . . . 102
PRESENT . . . 102
SIZE . . . 103
TESTOFF . . . 104
TESTON . . . 104
VALID . . . 105

Chapter 12. Logical functions . . . 107
ALL . . . 107
EITHER . . . 107
IF . . . 108
ISALPHA . . . 109
ISLOWER . . . 110
ISNUMBER . . . 110
ISUPPER . . . 111
NOT . . . 112
OR . . . 112
WHEN . . . 113

Chapter 13. Implementing a library EXIT function . . . 115
EXIT function's library interface . . . 115
Using the EXITPARAM Structure . . . 115

Chapter 14. Lookup and reference functions . . . 119

CHOOSE . . . 119
DBLOOKUP . . . 119
DBQUERY . . . 122
DDEQUERY . . . 125
EXTRACT . . . 125
GETANDSET . . . 126
GETDIRECTORY . . . 127
GETLOCALE . . . 128
GETFILENAME . . . 129
GETPARTITIONNAME . . . 130
GETRESOURCEALIAS . . . 130
GETRESOURCENAME . . . 131
GETTXINSTALLDIRECTORY . . . 131
INDEX . . . 132
INDEXABS . . . 133
LASTERRORCODE . . . 133
LASTERRORMSG . . . 134
LOOKUP . . . 135
MEMBER . . . 136
SEARCHDOWN . . . 137
SEARCHUP . . . 138
SORTDOWN . . . 139
SORTUP . . . 140
UNIQUE . . . 140

Chapter 15. Math and statistics functions . . . 143
ABS . . . 143
ACOSINE . . . 143
ASIN . . . 143
ATAN . . . 144
ATAN2 . . . 144
COSINE . . . 144
COSINEH . . . 144
COUNT . . . 144
COUNTABS . . . 145
EXP . . . 146
FACTORIAL . . . 146
FROMBASETEN . . . 146
INT . . . 147
LOG . . . 147
LOG10 . . . 147
MAX . . . 148
MIN . . . 148
MOD . . . 149
POWER . . . 149
RAND . . . 149
ROUND . . . 150
SIN . . . 150
SINH . . . 150
SQRT . . . 151
SUM . . . 151
TAN . . . 151
TANH . . . 152
TOBASETEN . . . 152
TRUNCATE . . . 152

Chapter 16. Text functions . . . 153
BCDTOTEXT . . . 153
COUNTSTRING . . . 153
CPACKAGE . . . 154

CSERIESTOTEXT . . . 155
CTEXT . . . 156
DATETOTEXT . . . 156
FILLLEFT . . . 157
FILLRIGHT . . . 158
FIND . . . 159
HEXTEXTTOSTREAM . . . 160
LEAVEALPHA . . . 161
LEAVEALPHANUM . . . 161
LEAVENUM . . . 162
LEAVEPRINT . . . 162
LEFT . . . 163
LOWERCASE . . . 164
MAX . . . 164
MID . . . 165
MIN . . . 165
NUMBERTOTEXT . . . 166
PACKAGE . . . 167
PROPERCASE . . . 167
REVERSEBYTE . . . 168
RIGHT . . . 168
SERIESTOTEXT . . . 169
SQUEEZE . . . 170
SUBSTITUTE . . . 171
TEXT . . . 172
TEXTTOBCD . . . 173
TEXTTONUMBER . . . 173
TEXTTOTIME . . . 174
TIMETOTEXT . . . 175
TODATETIME . . . 176
TONUMBER . . . 176
TRIMLEFT . . . 178
TRIMRIGHT . . . 179
UPPERCASE . . . 180
WORD . . . 180

Chapter 17. XML functions . . . 183
VALIDATE . . . 183
XPATH . . . 183
XSLT . . . 184

Chapter 18. Custom functions . . . 185
Creating a custom function . . . 185

Chapter 19. Date and time format strings . . . 189
Time units . . . 189
Date units . . . 189
Binary date and time format strings . . . 190
Japanese date and time format strings . . . 190
Japanese date format strings . . . 191
Japanese time format strings . . . 191
Western date and time format strings . . . 192
Western date format strings . . . 192
Western time format strings . . . 193

Chapter 20. Number format strings . . . 195
Leading sign format strings . . . 195
Trailing sign format strings . . . 195
Substring format strings . . . 195
Whole number and fraction format strings . . . 196

Chapter 21. RUN function return codes 197

Chapter 22. Character set codes for CPACKAGE, CSERIESTOTEXT, and CTEXT . . . 201

Notices 209
Programming interface information 211
Trademarks and service marks. 211

Index 213

Chapter 1. Expressions and evaluations

You can use functions and expressions to create component rules in the Type Designer and to create map rules in the Map Designer.

The **Examples** directory, located in the product installation directory, contains type trees and maps that can help you learn how to accomplish your mapping tasks. These type trees and maps contain component and map rules that show the use of many functions and expressions.

Expressions

An expression is a statement about data objects. Expressions are used in component rules in the Type Designer and in map rules in the Map Designer.

The following are examples of expressions:

```
Account Balance = Credits - Debits
TrunkRoom < 15
PRESENT (Store#:StoreInfo)
```

As you enter an expression, you can add spaces before and after component names and operators to make the expression more readable. Additional spaces have no effect on an expression.

Expressions are a combination of literals, object names, operators, functions, and map names.

Component rule expressions evaluate to true or false

Component rules are expressions that evaluate to ″true″ or ″false″. For example, the result of each of the following component rules is ″true″ or ″false″.

```
WHEN (PRESENT (Qualifier), PRESENT (Address))
COUNT (Exchange) < COUNT (Purchase)
Invoice#:Item = Invoice#:Item[1]
```

Map rule expressions evaluate to data

Map rules are expressions that evaluate to data.

For example, the result of each of the following map rules is data:

```
= "Florida"
= Price:Input
= COUNT ( Name:Roster )
= RecordMap ( FixedRecord )
```

Literals

A literal is a constant value. A literal may be a number or a text string.

The rules for using numeric literals are:

- A numeric literal is in integer or decimal format and can be signed.
- A comma separator should not be included in a numeric literal.
- A numeric literal cannot be greater than 254 digits.

The rules for using text literals are:

- A text literal is enclosed in double quotes.
- A double-quote included in a text literal is released by another double quote.

The following are some examples of literals:

```
"This is a text literal"
"ABC Company"
"Some ""quoted data"" to show use of a double double-quote."
1.23
-9
1045
```

Carriage return/line feeds cannot be within a quoted string. To use a carriage return/line feed the string must be broken into two strings with <CR><LF> between them. For example:

```
="username=?????" + "<CR><LF>" + "&password=?????"
```

will place the output username and password information on separate lines.

<CR><LF> must be within quotation marks.

Data object names

A data object is referenced in an expression by its name. An object name can be as simple as a card name or a local type name. It can also be complex depending on the object you want to reference.

Object names in map rules

In a map rule, data object names always refer to the card name that contains the data object name. So, in a map rule, a data object name ends with a card name. In the following example image, the object name for the component **Contact** of the card **ContactFile** is **Contact:ContactFile**.

The general definition of an object name in a map rule can be divided into three parts as described in the following table. If one of these parts is used, it must be sequenced according to the order in the table.

Parts of an Object Name
Required

Comment name
No

Set of component-paths, each ending with either a colon or the reserved word IN No

Card name
Yes

When IN is used, there must be a space before and after it, such as `Dependent IN Claim`.

Object names in component rules

In a component rule, data object names always refer to components in the same component list. In a component rule, a data object name ends with a component name. For example, if **LineItem** is in a component list, the object name for the **Qty Info** component of **LineItem** is **Qty Info:LineItem**.

In the preceding **Qty Info:LineItem** example, the colon would be interpreted as ″is a component of″. If the example was **Qty Info LineItem,** the space between **Info** and **LineItem** would be interpreted as ″is a subtype of″.

The general definition of a component rule object name can be divided into the two parts described in the following table. If one of these parts is used, it must be sequenced according to the order in the table.

Parts of an Object Name	Required
Set of one or more component-paths , infix separated with either a colon or the reserved word, **IN**	At least one is required
The reserved word, COMPONENT	No

IN and COMPONENT are reserved words. Reserved words are not case-sensitive.

Card name

A map rule can reference the object of an entire card. A card name is a simple type name from 1 to 32 characters in length. For example, if the input card name is **Invoice**, the name for the card object would be **Invoice**.

Local type name

An object name can be a local type name. A local type name is specified as one or more simple type names separated by spaces, such as:

```
City Field
```

Local type names are typically used to refer to a component. If used in a component rule, for example, the component referenced in a rule is specified as a local type name. In a map rule, local type names refer to components contained in a card object.

Local type names can also be used to refer to the partitioned type of a component. When a type is partitioned, you can refer to all of its partitions by simply referring to the partitioned type.

For this example, the type **Record** is partitioned into **Header**, **Detail**, and **Trailer**. The object name that references **Header**, **Detail**, and **Trailer Record** is: **Record**

Here is another example: the type **Transcript** has **Header Record**, **Detail Record**, and **Trailer Record** as components contained within it. The object name that references all **Header**, **Detail**, and **Trailer Records** within **Transcript** would be **Record IN Transcript**.

Partition list

A partition list is a set of partition names separated by the symbols <>. A partition name is the simple name that represents the partitions found under the parent, or partitioned, type. For example, the partition Header of the type Record would have this component path: **Header<>Record**

Component path

A component path is specified as an optional partition list, followed by a local type name and then followed by an optional index. Here are some examples of component paths:

```
Grand<>Ball Room[5]
Ball Room[LAST]
Really<>Grand<>Ball Room
Ball Room
```

Indexed object names

An object name can refer to a particular occurrence of a data object. The index of the occurrence appears in square brackets immediately after its name. For example, the object name of the third **Note** would be:

```
Note[3]
```

The index between the square brackets can be an integer or the reserved word, ″LAST″ (a special index value that refers to the last data object of a particular series). The index cannot be another object name. To use a variable index, use the CHOOSE function.

LAST is interpreted in the context in which it is used.

Using LAST in a map rule to reference an input

In a map rule, when referencing an input, LAST refers to the very last occurrence of that input. For example, the last **Note** of the input **Report** would have the name:

```
Note[LAST]:Report
```

In this context, all of the input has been validated before map rules are evaluated.

Using LAST in a component rule

In a component rule, LAST refers to the last occurrence found. For example, in the rule:

```
PO# Field:Order = PO# Field:Order[LAST]
```

LAST refers to previous occurrence of **Order** because when this rule is evaluated, the last known **Order** is the last one found.

Using LAST in a map rule to reference an output

In a map rule, when referencing an output, LAST refers to the last built occurrence of that object. For example, using the same expression:

```
PO# Field:Order = PO# Field:Order[LAST]
```

LAST refers to the previous occurrence of **Order** because, when this rule is evaluated, the last known **Order** is the last one built.

LAST is a reserved word and is not case-sensitive.

Component paths separated by a colon

A component in the component list of another object is always referenced by a colon (**:**). The local type name of the nested component is followed by the colon. To the right of the colon is the name of the object that contains the component. For example, the **Line Item** component of **PO** would have this name:

```
Line Item:PO
```

The **Last Name** component of **Client**, which is in turn a component of **Account**, would have this name:

```
Last Name:Client:Account
```

Component paths separated by IN

To reference all occurrences of an object contained in another object, use the keyword IN.

For this example, **ShipSchedules** appears as a component of different components within **OrderAck**.

The object name for all occurrences of **ShipSchedules** within **ORD_ADR_OUT_WD Record** is:

```
ShipSchedules IN ORD_ADR_OUT_WD Record:OrderAck
```

When IN is used, the contained object name must refer to the same type. In the example, the **ShipSchedules** component of **ORD_ADR_OUT_WD Record** and the **ShipSchedules** component of **OrderLine Structure** must be of the same type.

In the Map Designer, you can refer to all occurrences of an object contained in a card object. The object name for all occurrences of **ShipSchedules** within **OrderAck** is:

```
ShipSchedules IN OrderAck
```

Referring to all occurrences with IN COMPONENT

In the Type Designer, you can refer to all occurrences of an object contained in a component list. To refer to all components in a component list, up to a given component, the word COMPONENT is used.

For example, the rule on the component **Conclusion Topic** needs to count all of the valid **Topics** up to, and including, **Conclusion Topic**.

The `IN COMPONENT` expression can be used:

```
#Topic Field:$ = COUNT (Topic IN COMPONENT)
```

The above component rule works only if **Topic** is partitioned.

Comment object name

A comment name can be either an in-comment name or an @-comment name.

An in-comment name references all comment objects within a specified object. An in-comment name is specified as a component-path, a space, the reserved word IN, followed by another space, followed by the component path of the component on which the floating component type is defined. For example, given the data structure shown below, to reference all **INPUT Parameters** for a particular **Input**, regardless of whether they followed **Method OpenFile** or **Method GetEntry**, the object name would be **INPUT Parameters IN Input**.

An @-comment name references all comment objects that follow immediately after a specified object. An @-comment name is specified as the component path of the comment, followed by the reserved symbol "@", followed by the component path of the component that follows the comment. For example, to reference only the

INPUT Parameters that follow immediately after the **Method OpenFile** (and before **Method GetEntry**), the object name would be **INPUT Parameters @ Method OpenFile:Input**.

Shorthand notation

In a rule, a dollar sign ($) can be used to refer to the object to the left of the rule cell. In the Type Designer, the $ sign refers to the object you are qualifying. In the Map Designer, the $ refers to the object to which the rule evaluates.

In certain cases, you can also use a period between colons to shorten the reference to a data object name. The ″Use Ellipses″ command can be used to automatically shorten data object names you drag into rule expressions.

Using the dollar sign ($)

The dollar sign can be used to refer to the object to the left of a rule cell.

For example, the component rule below applies to the component **RunRule**.

The **RunRule** in the rule can be replaced by **$**. You can remember it this way-a dollar sign always means ″whatever component appears to the left of the rule″.

Here is an example of a Map Designer rule that uses the dollar sign:

```
Order:Invoice = MapMyOrder ( Order:PO , Index ( $ ) )
```

In this example, each time this rule is evaluated, the index of the **Order:Invoice** being built is the object referred to by the **$**.

Using ellipses

A single period can be used, like ellipses, in place of object names, as long as that object name can be interpreted as a unique object.

For example, if you have the following object name:

```
City Field:BillTo Customer:Account:File
```

you may be able to replace it with

```
City Field:.:File
```

If file also contains another **City Field**, such as

```
City Field:ShipTo Customer:Account:File
```

and you use the **Use Ellipses** option, the system returns

```
City Field:BillTo Customer:.:File
```

Evaluating expressions

Expressions are evaluated based on the context in which data object names, literals, operators, function names and map names appear in the expression, and the data on which it operates. A component rule always evaluates once for each occurrence of a component. A map rule can evaluate many times.

When a map rule evaluates, the Map Designer selects an evaluation set of values and then evaluates the rule. Then it selects another evaluation set of values, if there is one, and evaluates the rule again. This continues until all the different evaluation sets have been used.

The same number of evaluations is always produced, but output objects can be ordered differently, depending on how you define your map.

For a selected evaluation set, a particular instance of a rule is evaluated using parentheses, operator precedence rules, and left-to-right precedence rules to get one result.

Card order can influence the order of evaluation sets

When an evaluation set is selected, card object names are ordered by their position in the map: card 1 before card 2, and so on. The card order for a map can affect the order of output results.

For example, suppose you want to evaluate the following map rule.

```
NumberSet ( s ) = B:Card2 + A:Card1
```

In this example, an evaluation set for this rule consists of one value for B and one value for A. One evaluation set produces one **NumberSet**.

As an example, here there are three As and three Bs in the data:

A Values	B Values
1	3
2	2
3	1

There are nine evaluation sets. The order of the cards may affect the order of the evaluation sets.

If Card 1 is the first card, you get the following results:

Evaluation #	A	B	Result
1	1	3	4
2	2	3	5
3	3	3	6
4	1	2	3
5	2	2	4
6	3	2	5
7	1	1	2
8	2	1	3
9	3	1	4

If Card 2 is the first card, you get the following results:

Evaluation #	B	A	Result
1	3	1	4
2	2	1	3
3	1	1	2
4	3	2	5
5	2	2	4
6	1	2	3

If Card 2 is the first card, you get the following results:

Evaluation #	B	A	Result
7	3	3	6
8	2	3	5
9	1	3	4

If both A and B are contained in the same card object, evaluation sets are selected based on the order A and B appear in the rule. Leftmost objects are selected first. If you change the rule to this:

```
NumberSet ( s ) = B:Card2 + A:Card2
```

the same results are produced as when card 2 is the first card.

Functions influence the number of evaluation sets

For this example, you have the following rule:

```
NumberSet ( s ) = A:Card1 + SUM ( B:Card1 )
```

Using the same values for A and B, there are only three evaluation sets as shown.

NumberSet[1] = 7

A Values	B Values
1	**3**
2	**2**
3	**1**

NumberSet[2] = 8

A Values	B Values
1	**3**
2	**2**
3	**1**

NumberSet[3] = 9

A Values	B Values
1	**3**
2	**2**
3	**1**

If your rule looks like:

```
NumberSet ( s ) = SUM ( A:Card1 ) + SUM ( B:Card1 )
```

there is one evaluation set: NumberSet[1] = 12

A Values	B Values
1	3
2	2
3	1

If you have the following rule:

```
NumberSet ( s ) = A:Card1 + EXTRACT ( B:Card2 , B:Card2 != 2 )
```

the order of the operands does not affect the order of evaluation set selected. This is because an A is selected, then the EXTRACT is evaluated, then evaluation sets are determined for that A and a given EXTRACT result. When all EXTRACT results are used, the next A is selected, and so on.

Given the same data, this time you get the following results:

Evaluation #	A	B Extract Values	Result
1	1	3	4
2	1	1	2
3	2	3	5
4	2	1	3
5	3	3	6
6	3	1	4

As another example, the following map rule applies to the output Status(s):

```
= IF ( Quantity:Order Record:Order > 10 &
OR ( Store:Order Record:Order = "A" ) ,
      "Accept" , "Reject" )
```

The first argument of the IF function is an expression. One evaluation set of this expression requires:

- the **Order**
- one **Order Record**
- one **Quantity** of the selected **Order Record**
- all the **Stores** of the selected **Order Record**

The IF function evaluates once for each **Quantity** of each **Order Record**. Other than the first argument of the IF function, nothing else in the map rule affects the number of times the map rule will be evaluated. So, if there are 1000 **Quantity** data objects contained in **Order**, this map rule will evaluate 1000 times.

Object names influence the number of evaluation sets

In some expressions, the same data object name may appear more than once. The same data object name always refers to the same data object. The same data object name influences the combinations of evaluation sets that are selected. For example:

```
BackOrder(s) = Qty Ordered:Record:Order - Qty Received:Record:Order
```

When a **BackOrder** evaluation set is selected, the **Record** that contains **Qty Ordered** is always the same **Record** that contains **Qty Received**. And an **Order** is always the same one for one evaluation set. When the same object name is used more than once in the same expression, both references bind to the same object.

If there is always one **Qty Ordered** per **Record** and one **Qty Received** per **Record,** the following would be sample data:

Record #	Qty Ordered	Qty Received
1	5	4

Record #	Qty Ordered	Qty Received
2	4	4
3	6	2

This would produce three **Back Orders**, one for each **Record**.

You can coordinate nested data objects easily by the way an object name is used.

Using object names to coordinate evaluation sets

In the following example, common object names are used to coordinate **Qty** and **UnitPrice** so they do not get mixed up across different **Records**:

```
Extension = SUM ( Qty:Detail Record:Order * UnitPrice:Detail Record:Order )
```

This rule has only one evaluation set because the SUM consumes all the objects referenced in the rule.

Using IN to decouple object name coordination

Suppose you have a situation in which you do not want to coordinate common objects. You may, for example, want to count different objects contained in the same object, as in:

```
Summary = COUNTabs ( Header:Order:Message ) + COUNTabs ( Line Item:Order:Message )
```

If there are multiple **Orders** in your data, the **Summary** would not be evaluated correctly. This rule would only count the **Headers** of the first **Order** and add it to the count of all Line Items of the first **Order**. To count all the **Headers** and all the **Line Items** in all the **Orders**, use IN:

```
Summary = COUNTabs ( Header IN Message) + COUNTabs ( Line Item IN Message )
```

Using IN to decouple different partitions in the same rule

If data objects in an expression are different partitions of the same type, the expression may evaluate to ″none″.

For example:

```
MyMap ( Good<>Design , Best<>Design )
```

Good and **Best** are partitions of the same type, **Design**. A given **Design** could never be both a **Good<>Design** and a **Best<>Design**; therefore, the second argument of **MyMap**, **Best<>Design**, evaluates to ″none″.

To reference both the **Good** and **Best** partitions, use IN:

```
MyMap ( Good IN Design , Best IN Design )
```

Operators

Operators are used for arithmetic functions and text functions on items. If an operator requires two operands, the operator symbol appears between them. For example, in the expression **a + b**, the operator symbol ″**+**″ appears between the operands **a** and **b**.

Operators are reserved symbols that cannot be used in type names.

Arithmetic operators

Operator	Description
+	addition
-	subtraction
*	multiplication
/	division

Text operators

Operator	Description
+	concatenation

Logical operators

Logical operators are used in expressions that evaluate to ″true″ or ″false″.

Operator	Description
^	exclusive or
&	and
|	or
!	not
=	equals

Comparison operators

Operator	Description
=	equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
!=	not equal to
NOT=	alternative for not equal to

The results of comparison operators are based on the native collating sequence of the machine used to run the map.

Order of operator evaluation

Use parentheses to group operations. Expressions within parentheses are evaluated before performing other operations. For example:

```
(UnitPrice - Discount) * Tax
```

The expression **UnitPrice - Discount** is evaluated before the rest of the expression.

Some operators are evaluated before others according to standard rules of precedence. Operators are evaluated in the following order:

Precedence	Operators	Description
first	( )	Operations within parentheses
second	+ and -	unary plus and minus The sign indicates a positive or negative expression, for example: - quantity*rate
third	=, <, >, <=, >=, !=	comparative operators
fourth	^	logical operator ″exclusive or″
fifth	&	logical operator ″and″
sixth	|	logical operator ″or″
seventh	* and /	multiplication and division
eighth	+ and -	addition and subtraction

Operators of equal precedence are evaluated left to right.

Operands

Operands are the arguments for an operator. For example, in the expression **a + b**, the **+** is the operator symbol, and **a** and **b** are the operands.

An operand can be any of the following:

- a literal
- a data object name
- a function
- another operator

Using functions in expressions

Functions perform a particular action on its input arguments. A function is written like this:

```
FUNCTION (argument1, argument2, ... argumentn)
```

The arguments of the function appear inside the parentheses, separated by commas. A function may require a fixed number of arguments or may allow a varying number of arguments. You can use functions anywhere in an expression.

Function arguments

Input arguments themselves can be expressions. Some functions limit the type of expression that can be used as an input argument. The syntax specification tells you the number of data objects that can be used for one function evaluation.

There is *always* one output argument for a function. The specification of the output argument tells you: 1) the type of the result produced by a function and 2) the number of objects that can be produced when the function evaluates once.

You need to know the type of the output argument because functions can be used as other arguments. The result of a function can also be used to directly produce an output data object contained in an output destination.

When the Map Designer analyzes the interfaces of expressions in rules during the build process, it ensures that the output argument of a function matches the input argument it is used for. The analysis also checks the type of the output argument when that function is used to directly produce an output data object.

The output argument of a function specifies the result that the function produces for one evaluation. For example, one evaluation of the ABS function produces one positive number. One function evaluation may produce multiple outputs. If it does, the expression that contains that function may produce multiple evaluation sets.

Input arguments

The input arguments for a function specify the information the function uses to return a result. Some functions require an exact number of input arguments. For other functions, the number of input arguments can vary. Optional arguments for a function are shown in brackets [ ].

To correctly use a function, you must use the correct arguments in the correct order.

Each argument can, itself, be an expression. In general, an input argument may be any of the following:

- an object name
- a literal
- a function-name and its arguments
- an operator and its arguments
- an enumerated series of literals, such as {a, b, c}
- a map name and its arguments

For example, some valid expressions that use the ABS function are:

```
ABS ( Quantity:LineItem )
ABS ( UNIQUE ( Quantity:LineItem )
ABS ( Quantity:LineItem - 300 )
```

Each function has its own expression syntax for its input arguments. For example, the ABS function cannot use a map as an argument. See "Syntax of a Function" for notation used to specify valid expression syntax.

Nested input arguments

When a function, operator, or map is used as an input argument, each of these has arguments of its own. Arguments of other arguments are said to be *nested*. For example, in the use of the ABS function, you can use a function as an argument:

```
ABS ( UNIQUE ( Quantity:LineItem ) )
```

The output of the UNIQUE function becomes the input argument to the ABS function.

Output arguments

The output argument of a function indicates the type of object the function produces and the number of objects a function can produce. An output argument is specified as one of the following:

- A single data object of a type
- A series of data objects of the same type
- ″True″ or ″False″

  If the output of a function is a series, that function may be used only in a map rule in the Map Designer. It cannot be used in a component rule in the Type Designer. For example, you cannot use the CLONE, EXTRACT, REJECT, SORTDOWN, SORTUP or UNIQUE functions in the Type Designer.

Function arguments and evaluation

When a function is evaluated, the number of argument objects used for one evaluation depends on the function. In the specification of the syntax for a function, the number of input objects that can be used for each argument for one function evaluation is expressed as ″single″ or ″series″.

Some functions use a single object as the value of an argument for one evaluation. For example, the ABS function uses a single object as the value of its argument for one evaluation.

```
ABS ( Quantity:LineItem )
```

When a single object is used for one evaluation, the function itself can evaluate many times if there is more than one data object that fits the argument definition. For example, if there were ten **Quantity:LineItem** data objects, the ABS function could evaluate ten times.

Some functions use an entire series of objects as the value of an argument for one evaluation. For example, the COUNT function uses an entire series of data objects as the value of its argument for one evaluation.

```
COUNT ( LineItem:PO )
```

When a series of data objects is used for one evaluation, the function evaluates just once when there is more than one data object that fits the argument definition. For example, if there are ten **LineItem:PO** data objects, the COUNT function would evaluate only once.

Some functions use an entire series as input and produce a series as output. For example, the evaluation set for the EXTRACT function produces a series from an input series. When a function produces a series, each output can be selected for different evaluation sets of the expression that contains that function.

Consider the following expression:

```
Line Item(s) = EXTRACT ( Line Item:Order, Qty:Line Item:Order > 1000 )
```

Each **Line Item** that fits the specified criteria produces a **Line Item** of **Order**. In this example, the map rule is evaluated many times, once for each **Line Item** produced by the EXTRACT function.

When functions are part of other expressions, the number of evaluation sets for that function depends on the object names used in the entire expression. For example:

```
Debit ( s ) = ABS ( Debit:Account:Input )
```

There may be many evaluation sets for the above expression. The ABS function may be evaluated many times, once for each **Debit** of each **Account** of **Input**. However, if the ABS function is part of a more complex expression, common objects of the expression may determine the number of ABS evaluations.

Map names in expressions

A map may be used like a function and may be referenced in a map rule. This kind of map is called a functional map. The syntax of a functional map in an expression is the same as the syntax of a function:

```
FunctionalMapName ( argument1 , argument2 , ... argumentn )
```

Note: Unlike a function, a map cannot be used as an argument to an operator or another map. A map may be used as an argument to a function only if the output of the map evaluates directly to output data objects. For example,

Valid:

IF (A > 10 , BigMap (X) , LittleMap (Y))

Not valid:

IF (FunnyMap (A) , ″2″ , ″3″)

Evaluation of functional maps in an expression

A single evaluation of a functional map requires one object for each input argument. The result of the evaluation is always one output object. As with functions and operators, when a functional map is used in an expression, evaluation sets for that expression may cause the functional map to be evaluated many times.

Consider the following expression:

```
MakeForm ( EntryForm:Input , GroupInfo:Input )
```

If there are two occurrences of **GroupInfo** and four occurrences of **EntryForm**, the map **MakeForm** will be evaluated once per combination of **EntryForm** and **GroupInfo**-a total of (2 x 4) = 8 times.

Consider another expression:

```
Fmap ( a:Input , b:Input , c:Input )
```

The objects **a**, **b**, and **c** have only **Input** in common. The expression that contains the **Fmap** reference will be evaluated once per combination of **a**, **b**, and **c**.

To illustrate how the functional map will be evaluated, use a very simple data file of **a**'s, **b**'s, and **c**'s. Each time the map is evaluated, one **a** object, one **b** object, and one **c** object are selected. The first time the map is evaluated, the data objects are a1, b1, and c1. The next time it is evaluated, the data objects are a2, b1, and c1, and so on.

An important difference in how the Map Designer evaluates a functional map in an expression is when an input argument evaluates to ″none″.

See ″When an Input Argument of a Function Evaluates to NONE″ .

Expressions that evaluate to NONE

The following discussion explains how the Map Designer evaluates an expression when expressions within it evaluate to ″none″.

When an operand evaluates to NONE

The following table explains how an operator expression evaluates when an operand evaluates to ″none″.

The symbol **A** represents one operand of the expression and **B** represents the other operand. The symbol **!=** means ″not equal to″.

Operator Expression	Condition	Evaluates to:
A / B	A = NONE B = NONE	NONE
A*B	A = NONE B = NONE	NONE
A / B	A != NONE B = NONE	0
A*B	A != NONE B = NONE	0
A + B	A = NONE B = NONE	NONE
A - B	A = NONE B = NONE	NONE
A + B	A = NONE B != NONE	B
A - B	A = NONE B != NONE	-B
A + B	A != NONE B = NONE	A
A - B	A != NONE B = NONE	A
A > B	A = NONE B = NONE	FALSE
A < B	A = NONE B = NONE	FALSE
A = B	A = NONE B = NONE	TRUE
A > B	A = NONE B != NONE	FALSE
A < B	A = NONE B != NONE	TRUE
A = B	A = NONE B != NONE	FALSE
A > B	A != NONE B = NONE	TRUE
A < B	A != NONE B = NONE	FALSE
A = B	A != NONE B = NONE	FALSE

For example, in this operator expression:

```
Total = #Guests + 1
```

if **#Guests** evaluates to ″none″, the value for **Total** will be 1.

When an input argument of a function evaluates to NONE

If any input argument of a function evaluates to ″none″, the function may not evaluate to ″none″. For example, ABS (NONE)=NONE, but COUNT (NONE)=0.

When an input argument of a functional map evaluates to NONE

If any argument of a functional map evaluates to ″none″, the functional map will not be evaluated for that evaluation set.

For example:

```
MakeForm ( EntryForm:Input , GroupInfo:Input )
```

Where the object **GroupInfo** is optional; it has a component range of (0:3). If there are no occurrences of **GroupInfo** in the **Input** data object, the map **MakeForm** will not be evaluated at all.

Here is another example:

```
OrderMap ( OrderRecord:Order:Input , SummaryRecord:Order:Input )
```

If **SummaryRecord** has a component range of (0:1) and it is missing in a particular **Order**. The map **OrderMap** will not be evaluated for that **Order**.

If you want **OrderMap** to be evaluated even when **SummaryRecord** is missing, use the entire **Order** as an input argument, rather than **SummaryRecord**. This will ensure that **OrderMap** will be evaluated, even if **SummaryRecord** is missing. Then, in the map **OrderMap**, you can still map from **SummaryRecord**.

The rule you should use is this:

```
OrderMap ( OrderRecord:Order:Input , Order:Input )
```

When an input of an executable map evaluates to NONE

The evaluation of a map referenced in an expression differs from the evaluation of that map when it is executed as a functional map and the evaluation of a function because of how the NONE is treated.

The evaluation of a map used as a function is different from the evaluation of that map when it is run as an executable map. When a map is run as an executable, if any input card data object has no content, the map will still run. This is not true when that map is referenced as a function. If any input argument of a functional map evaluates to ″none″, the map will not evaluate.

Impact of track setting on order of output

This property setting determines whether objects are tracked according to content or according to their position within a series. This setting will determine the sequence in which those objects are evaluated when referenced.

For example, assume that **RunsByInning** is a component of **BallGame**. **BallGame** is defined as an explicit format, delimited (by a colon) group. The data for **BallGame** is as follows:

```
2::1:1::1::3
```

- **Track Content**. If **BallGame** has a **Track** setting of **Content**, all instances of **RunsByInning** with content will be mapped before any empty instances. So, if **BallGame** was mapped to itself, the output would be:

  ```
  2:1:1:1:3
  ```

- **Track Places**. If **BallGame** has a **Track** setting of Places, instances of **RunsByInning** will be mapped in the sequence in which they occur, including those instances that are empty. So, if **BallGame** was mapped to itself, the output would match the input.

  ```
  2::1:1::1::3
  ```

  Track Places will only have an impact on the sequence of the output when *both* the input group object and output group object are defined as **Track Places**.

However, **Track Places** will have an impact in situations in which an input defined with **Track Places** is referenced by index. For instance, in the above example,

- If **RunsByInning** was defined as Track Places, **RunsByInning[2]:BallGame** would have a value of **2** because the value in the second ″place″ in **BallGame** is a **2**.
- On the other hand, if **RunsByInning** was defined as **Track Content**, **RunsByInning[2]:BallGame** would have a value of **1** because the second instance of **BallGame** with content has a value of **1**.

Evaluated expressions assigned to output items

NONE assigned to an output number

When an output item is defined as a number, its value is determined from the result of the evaluated expression that is then converted to its output form as follows:

Output is specified as:	Evaluated value	Output value
Required	0	0
Required	NONE	0
Optional	0	0
Optional	NONE	NONE

Automatic item format conversions

When an item is assigned an output value, the Map Designer will automatically convert an item with the same interpretation (either number, text, date, or time) from one presentation to another. For example, if you want to map an item that is a number and whose presentation is integer to an item that is also a number but whose presentation is decimal, the Map Designer will automatically convert it.

The Map Designer fills in missing information in date and time conversions in the following way:

- **Dates with a year, but no century**. The century will be determined using the **CenturyDerivation** map setting.
- **Missing part of an output date format**. If a portion of the format of an output date cannot be derived from the input date, the missing part will have pound characters (#) in its place. For example, if `09/98` is the date for an input with a format string of MM/YY and that date was mapped to an output date with a format of MM/DD/YY, the output date would be built as **09/##/98**.
- **Missing part of an output time format**. If a portion of the format of an output time cannot be derived from the input time, the missing part will have zeroes (0) in its place. For example, if `12:14` is the data for an input with a format string of

HH12:MM and that time was mapped to an output time with a format of HH:MM:SS, the output date would be built as **12:14:00**.

To convert a number to text, a date or time, or vice versa, you must use a function to perform the conversion. For example, to convert a date to text, use the function DATETOTEXT.

Numeric precision

All numbers in the input and numeric literals in map rules are converted to multiple precision and converted to the format required at output time.

- If the multiple precision number does not overflow but does not fit into the maximum size of a numeric item, that item is filled with the number sign character (#) up to its maximum size.
- An arithmetic overflow is recognized when an arithmetic expression includes:
  - Divide by zero
  - A multiple precision number greater than 1.0E254

Arithmetic overflow is designated in the output object with up to a maximum of ten number sign (#) characters.

Chapter 2. Using functions

A function is an expression that generates an output by performing a certain operation on one or more inputs. Most functions can be used in both component rules and map rules. Those that produce a series can only be used in map rules.

Functions in a component rule

For example, the object **TotalQty** in your data is the sum of all the **Qty** objects in your data; in a component rule for **TotalQty**, you can use the SUM function to verify this relationship.

Functions in a map rule

Suppose you want to map your data differently according to the presence of a certain input data object. In your map rule, to check the presence of the data object, use the function PRESENT. To create the output according to whether the input data was present, use the function IF. Examples of the IF function can be found in ″Logical Functions″. Examples of the PRESENT function can be found in ″Inspection Functions″.

Chapter 3. Syntax of a function

The following is an example of a function:

```
FUNCTION ( argument1, argument2, ... argumentn )
```

When you use a function, substitute the name of a specific function for FUNCTION and substitute a specific expression for each input argument. Each input argument is separated by a comma and the set of arguments is surrounded by parentheses.

Chapter 4. Function argument syntax

Arguments typically have some restrictions on the expression used for that argument. For example, the input argument to the INDEX function can only be an object name. Depending on the function, one or many data objects can be used for an input argument for one function evaluation.

For each function listed in this reference guide, the input argument has a syntax definition that indicates:

- the number of objects that can be used for each argument for one function evaluation, expressed as either ″single″ or ″series″

- followed by -

- the type of expression that can be used for each argument.

For the output argument, the syntax specification is similar to the input argument syntax, except that the output is always used as a specific type of object, not an expression.

For example, single-item-expression means one evaluation operates on a single item that can be defined as an item expression. Series-item-expression means one evaluation operates on one or more input argument items that can be defined as an item expression.

In this reference guide, the terms listed in the following table are used to describe the syntax of function arguments.

Term Explanation

general-expression
Any valid combination of object names, literals, operators, function names, and map names.

condition-expression
Any valid combinations of object names, literals, conditional operators, and functions whose output arguments are specified as ″true″ or ″false″.

item-expression
Any valid combination of object names, literals, operators, and functions whose output argument is specified as an item.

If an item must be interpreted as a date, time, or number, the item is referenced by its interpretation. For example, if an item-expression must be text, it is specified as text-expression.

object-expression
An object name or a series producing function.

If an object expression is further limited to be an item, it is specified as item-object-expression.

object-name
The name of a data object, as defined in ″Data Object Names″. If an object name is further limited, it is specified as a prefix. For example, simple-object-name refers to a simple type name.

... Indicates that the function can take on any number of additional similar arguments.

[ ] Indicates that the argument is optional.

{ literal , literal ...}
An enumerated series of literals.

The syntax specification shows each input argument, using terms as described in the previous table, preceded by the word ″single″ or ″series″. For example:

Syntax:
ALL (series-condition-expression)

The arguments are defined using a name that describes its use by the function. A simple example is:

Meaning:
DATETONUMBER (date_to_convert)

Function names are shown in uppercase letters in this guide. However, function names are not case-sensitive.

There is also a description of what each function returns.

Returns:
A single integer

Chapter 5. General functions

CLONE

The CLONE function creates a specified number of copies of some object.

This function can be useful when the number of output objects to be built depends on a data value, rather than the number of objects that exist in the data.

Syntax:
CLONE (single-object-name , single-integer-expression)

Meaning:
CLONE (object_to_copy , number_of_copies)

Returns:
A series-object

The CLONE function returns a series of the object specified by *object_to_copy.* The output series consists of as many copies of the object as specified by *number_of_copies*. The value of each member of the resulting output series is the same as *object_to_copy*.

Examples

- LineItem Segment (s) = LotsOf (CLONE (Detail Row , Quantity:Detail Row))

  In this example, the input contains an object named **Detail Row** that has a **Quantity** item component. You want to create multiple **Line Item Segments** for every **Detail Row** in your input based on the value of **Quantity**. So, if **Quantity** has the value 5, you want to build five **Line Item Segments** for that **Detail Row**.

DEFAULT

The DEFAULT function allows a pre-defined value to be introduced into an expression.

DEFAULT, which is typically used with the EITHER function, returns the value assigned as the default value when the type was defined.

.

Syntax:
DEFAULT(single-type-expression)

Meaning:
DEFAULT(type_whose_defined_default_value_is_desired)

Returns:
A single-object

Examples

- DEFAULT(ZipCode)

  In this example, the defined default value for ZipCode is returned. When used with EITHER, DEFAULT might appear like this:

  EITHER(ZipCode,DEFAULT(ZipCode))

If the first argument evaluates to a value other than ″none″, that value is used. If the first argument evaluates to ″none″, the default value for **ZipCode** is used.

ECHOIN

The ECHOIN function returns a command (or property) to be used in a RUN function argument.

Syntax:
ECHOIN (single-integer-expression, single-text-expression)

Meaning:
ECHOIN (card#, item_or_group_interpreted_as_text)

Returns:
A single object

Examples

Before ECHOIN was introduced, the RUN function was used in this way:

```
RUN ("Mymap", "-IE1S"+NUMBERTOTEXT (SIZE(Data))+ " " + TEXT(Data)))
```

You can use ECHOIN with RUN, similar to the following:

```
RUN ("Mymap", ECHOIN(1,(Data)))
```

Related functions

HANDLEIN

HANDLEIN

The HANDLEIN function returns a command (or property) to be used in a RUN function argument.

You can use the HANDLEIN function when deriving large amounts of data from a parent map for use in a RUN map. The HANDLEIN function works by using the same instance of the parent map's data set in the RUN map, instead of duplicating it as the ECHOIN function does.

The result of the HANDLEIN function is a text object that defines the characteristics of the data to be used. This includes an internal handle, offset, and length of the data. The result of this can be seen in an execution audit log:

```
<ExecutionSummary MapStatus="Valid" mapreturn="0" ElapsedSec="0.1803"
BurstRestartCount="0">
   <Message>Map completed successfully</Message>
   <CommandLine>install_dir\sdq20.mmc -IH1 4:512:1024 -ae</CommandLine>
   <ObjectsFound>85</ObjectsFound>
   <ObjectsBuilt>23</ObjectsBuilt>
<SourceReport   card="1"   adapter="Handle"   bytes="1024"   adapterreturn="0">
   <Message>Success</Message>
   <Settings>4:512:1024</Settings>
   <TimeStamp>05:06:07 January 27, 2004</TimeStamp>
</SourceReport>
```

You cannot use a GETANDSET function against an input that is passed using HANDLEIN.

Syntax:

HANDLEIN (single-integer-expression, single-text-expression)

Meaning:

HANDLEIN (card_to_override, data_object)

Returns:

A character text item

When the HANDLEIN function returns a character text item, the string is prefixed with: **IH***x*, where *x* is the first parameter.

String format:

-IH*x* <*internal-handle*>.<*offset*>.<*length*>

Example:

-IH1 4.4096.512

A *single-integer-expression* is the number item that specifies the card number to override. A *single-text-expression* specifies the data object to use for the override.

In a RUN map rule, you can override a source card with a handle of a current map's card by specifying the card to override and the object to override it with.

Examples

In the following rule, the data is object c:b:a for input card 2 of the RUN map:

HANDLEIN(2, c:b:a)

The second parameter can be more complex if needed. For example,

HANDLEIN(2, SUBSTITUE(c:b:a, ″a″, ″A″))

Any data in object **c:b:a** that contains a lower case ″a″ would change it to an upper case ″A″ for the call to the RUN map. In cases such as this, where the data cannot be expressed contiguously within the original card(s), a scratch file is used to hold the expression and the calling map's handle is that of the scratch file.

There is no correlation between the handle returned in the output of HANDLEIN and the actual card number of the second parameter.

Related functions

Generally, the ECHOIN function is used to return a command or property to be used in a RUN function argument. However, for large amounts of data, using the HANDLEIN function can be more efficient.

For best results, use the ECHOIN function for small quantities of data, such as less than 100K, and the HANDLEIN function for larger quantities of data of more than 100K.

REFORMAT

The REFORMAT function returns a type object that results from replacing the syntax of the input type with the syntax of the output type. The initiator and terminator of the output type are built. The content result rules are determined based on the Input type and the Output type. Groups and Items can be used with REFORMAT.

REFORMAT cannot be used as an argument to a function, operator, or functional map.

Syntax:

REFORMAT (single-object-expression)

Meaning:

REFORMAT (type-to-convert)

Returns:

A type object

REFORMAT returns a type object whose content matches the input type object, but whose syntax matches the output type object. The following table details the expected results based on the input and output.

Input Type	Output Type	Result
Group	Group	The content for each output component results from the content of the corresponding input group component content. For example, output component 1 is matched with input component 1, output component 2 is matched with input component 2, and so on. The components of the groups must be items, not another group. If there is no matching input component for an output component, the output component's required occurrences are built as ″none″S. If there is no matching output component for an input component, the data of that input component is ignored. When a component occurrence is built, the delimiter (if any) of the contained output group is built and an object of the component's output type is built from the corresponding input type using the REFORMAT algorithm.
Group	Item	The text of a corresponding input group content.
Item	Group	The input item content is applied as though it were the first component of the output group.
Item	Item	The input item content is applied to the output item content.

Examples

- MyXMLPurchaseOrder = REFORMAT(MyCOBOL_PurchaseOrder)

  This function is useful when the same type structure may come from different type trees, such as different versions of the same EDI, or when converting legacy formats to XML.

Chapter 6. Bit manipulation and testing functions

SETOFF

The SETOFF function sets to zero a specified bit in a binary number. You can use this function to manipulate (turn off) a bit in a binary number.

Syntax:
SETOFF (single-binary-number-expression, single-integer-expression)

Meaning:
SETOFF (binary_number_to_change, bit_to_turn_off)

Returns:
A single binary number

The SETOFF function uses *bit_to_turn_off* to specify the bit of *binary_number_to_change* to be set to the value 0. The result is a binary number item of the same size as *binary_number_to_change*.

The value of *bit_to_turn_off* represents the position of the single bit in *binary_number_to_change* to be set off. (Bits are numbered from left to right, with the leftmost bit being bit 1.) If *bit_to_turn_off* is less than one or greater than the number of bits of *binary_number_to_change*, SETOFF returns *binary_number_to_change* unchanged.

Examples

- SETOFF (A , 16)

  Assume **A** is the two-byte binary value of ″1″ (which is all zeros except for bit **16**). The binary representation of the value in **A** is 0001.

  The result is the two-byte binary value, ″0″
- SETOFF (A , 38)

  Results in the two-byte binary value ″1″, the original value of **A**, because bit **38** does not exist in **A**.

SETON

You can use the SETON function to ″turn on″ a specific bit in a binary number. SETON sets the specified bit in the number to ″1″.

Syntax:
SETON (single-binary-number-expression, single-integer-expression)

Meaning:
SETON (binary_number_to_change, bit_to_turn_on)

Returns:
A single binary number

SETON uses the value of *bit_to_turn_on* to specify the bit of *binary_number_to_change* that should be set to the value 1. The result is a binary number item of the same size as *binary_number_to_change*.

Bit_to_turn_on represents the position of the single bit in *binary_number_to_change* to be set on. (Bits are numbered from left to right, with the leftmost bit being bit 1.) If *bit_to_turn_on* is less than one or greater than the number of bits of *binary_number_to_change*, SETON returns *binary_number_to_change*, unchanged.

Examples

- SETON (A , 15)

  In this example, assume **A** is the two-byte binary value of 1 (which is all zeros except for bit 16). The binary representation of the value in **A** is 0001.

  The function returns the two-byte binary value, 0011, which is the decimal value of 3.
- SETON (A , 40)

  Returns the two-byte binary value of 1-the original value of A- because bit 40 does not exist in **A**.

TESTOFF

The TESTOFF function tests a specified bit in a binary number item to see whether it is off.

Syntax:

TESTOFF (single-binary-number-expression , single-integer-expression)

Meaning:
TESTOFF (binary_number_to_test , bit_to_test)

Returns:
"True" or "false"

The value of *bit_to_test* specifies which bit of *binary_number_to_test* should be tested for the value 0. If *bit_to_test* has the value 1, it refers to the leftmost bit of *binary_number_to_test*.

The TESTOFF function returns "true" if the specified bit is off and returns "false" if the specified bit is on.

If *bit_to_test* is less than one or greater than the number of bits of *binary_number_to_test*, TESTOFF returns "false".

Examples

- TESTOFF (A, 16)

  Assume **A** is the two-byte binary value of "1", which is all zeros except for bit **16**. The binary representation of the value in **A** is 0001.

  This example returns "false".
- TESTOFF (A, 20)

  Returns "false" because bit **20** does not exist.

TESTON

The TESTON function tests a specified bit in a binary number to see if it is on.

Syntax:

TESTON (single-binary-number-expression, single-integer-expression)

Meaning:
TESTON (binary_number_to_test ,bit_to_test)

Returns:
"True" or "false"

The value of *bit_to_test* specifies the bit of *binary_number_to_test* to test for the value 1. If *bit_to_test* has the value 1, it refers to the leftmost bit of *binary_number_to_test*.

The function returns "true" if the specified bit is on; it has the value 1. It returns "false" if the specified bit is off; it has the value 0.

If *bit_to_test* is less than one or greater than the number of bits of *binary_number_to_test*, TESTON returns "false".

Examples

- TESTON (A , 16)

  Assume **A** is the two-byte binary value of "1", which is all zeros except for bit **16**. The binary representation of the value in **A** is 0001.

  This example returns "true".
- TESTON (A , 20)

  Returns "false" because bit **20** does not exist in a two-byte value.

Chapter 7. Conversion functions

BASE64TOTEXT

Use the BASE64TOTEXT decoding function to convert previously encoded data from BASE64 back to the original text. The data is converted to the character set of the output object used. For example, if the output object is EBCDIC the output will be EBCDIC.

Syntax:

BASE64TOTEXT (single-object-expression)

Meaning:

BASE64TOTEXT (BASE64_object_to_convert)

Returns:

A single-text-item

This function receives a text object in Base64-encoded format, converts the object to text, and returns a single data object that represents the original text object that was encoded. If an error occurs during conversion, no data is returned.

Example

BASE64TOTEXT(″U29tZSBFeGFtcGxlIERhdGE=″)

Output: Some Example Data

BCDTOHEX

The BCDTOHEX function converts an item from binary coded decimal (BCD) format to hexadecimal format. The binary value that is returned always has the high-order byte first, regardless of the platform. This is useful for testing a specific bit position for a specific value.

.

Syntax:

BCDTOHEX (single-text-expression, single-integer-expression)

Meaning:

BCDTOHEX (BCD_item_to_convert, length_of_output)

Returns:

A single binary bytestream

When using the BCDTOHEX function, the input argument is converted from BCD format to its binary value.

Numbers in BCD format have two decimal digits in each byte. Each half-byte, therefore, can contain a binary value from 0000 (which represents the digit 0) through 1001 (which represents the digit 9). Based on this definition, the following applies:

- If any half-byte of the BCD number contains the binary values 1101 or 1111, that half-byte is ignored.

- If any half-byte contains the binary values 1010, 1011, 1100, or 1110, the output of the function is ″none″.

The length of the output argument is specified as the value of the second input argument. The length must be 1, 2, or 4. If it is any other value, the length is assumed to be 2.

Examples

- You can use BCDTOHEX to test a specific bit position for a specific value. For example, BCDTOHEX is useful when the BCD value represents flag bits. The conversion performed by BCDTOHEX results in a predictable location for the flag bits across platforms where the bytes may otherwise be reversed.
- TESTON (BCDTOHEX (ProcessIndicator, 2) 10)

  Tests for the x`0040′ bit in a two-byte result.

  Note: This same test, using BCDTOINT on a PC, tests the x`4000′ bit.

Related functions

- BCDTOTEXT
- BCDTOINT
- TEXTTOBCD

BCDTOINT

The BCDTOINT function converts an item from BCD (binary coded decimal) format to integer format. The binary value that is returned has a form native to the machine on which the map is being run.

Syntax:

BCDTOINT (single-text-expression , single-integer-expression)

Meaning:

BCDTOINT (BCD_item_to_convert , length_of_output)

Returns:

A single-binary-number

BCD_item_to_convert is converted from BCD (binary coded decimal) format to its integer value.

Numbers in BCD format have two decimal digits in each byte. Each half-byte, therefore, can contain a binary value from 0000 (which represents the digit 0) through 1001 (which represents the digit 9). Based on this definition, the following applies:

- If any half-byte of the BCD number contains the binary values 1101 or 1111, that half-byte is ignored.
- If any half-byte contains the binary values 1010, 1011, 1100, or 1110, the output of the function is ″none″.

The length of the output binary number is specified by *length_of_output*. The length must be 1, 2, or 4. If it is any other value, the length is assumed to be 2.

BCDTOINT returns a binary value of the form native to the machine on which the map is being run. The value can therefore be used in arithmetic operations or compared to other numeric values. It should *not* be used for bit manipulations,

because the order of the bytes in the number is dependent on the platform on which the map is being run. For example, on a personal computer, the low-order byte is first, while on a mainframe, the high-order byte is first.

Examples

- BCDTOINT (bcdAmount Field , 2)

  If the **bcdAmount Field** contains **x'37'** in BCD format, this example returns the integer value of 37.

Related functions

- BCDTOTEXT
- BCDTOHEX
- TEXTTOBCD

BCDTOTEXT

The BCDTOTEXT function converts the digits in a BCD (Binary Coded Decimal) item to a text item containing the digits of the BCD-encoded item as a string of characters.

Syntax:

BCDTOTEXT (single-text-expression)

Meaning:

BCDTOTEXT (BCD_item_to_convert)

Returns:

A single text item

BCD_item_to_convert is converted from BCD format to a text string containing the digits of the BCD-encoded value as a string of characters.

Numbers in BCD format have two decimal digits in each byte. Each half-byte, therefore, can contain a binary value from 0000 (which represents the digit 0) through 1001, which represents the digit 9). Based on this definition, the following applies:

- If any half-byte of the BCD number contains the binary value 1101 or 1111, that half-byte is ignored.
- If the BCD item contains the binary value 1010, 1011, 1100, or 1110, the output of the function is ″none″.

Examples

- BCDTOTEXT (Qty:Item)

  If **Qty** is x`1234', the result is 1234.
- BCDTOTEXT (DiscountAmt)

  If **DiscountAmt** is x`0123', the result is 0123.
- BCDTOTEXT (TotalDollars)

  If **Total** is x`F123', the result is 123.

Related functions

- BCDTOHEX
- BCDTOINT
- TEXTTOBCD

CONVERT

The CONVERT function replaces each byte of a byte stream or text expression with a byte from another byte stream or text expression. The decimal value of each byte of the first argument is used to locate the corresponding byte in the second argument that will replace it.

Syntax:

CONVERT (single-byte-stream-or-text-expression, single-byte-stream-or-text-expression)

Meaning:

CONVERT (bytes_to_replace , replacement_bytes)

Returns:

A single byte stream or text item

The CONVERT function replaces each byte of *bytes_to_replace* with a byte from *replacement_bytes*. The byte chosen from *replacement_bytes* is the one whose index is the decimal value of *bytes_to_replace*. The first byte of *replacement_bytes* has the index value of zero. If there is no corresponding byte in *replacement_bytes*, CONVERT returns ″none″.

You can use the CONVERT function to convert ASCII to EBCDIC or EBCDIC to ASCII data. See the **convert.mms** map source file in the **examples\general\portdata** folder of the product installation directory. One of the maps in **convert.mms** is **ASCII_TO_EBCDIC**, which converts ASCII to EBCDIC. The **EBCDIC_TO_ASCII** map converts EBCDIC to ASCII.

ASCII to EBCDIC conversion can be performed automatically by defining the appropriate data language (ASCII or EBCDIC) for each data item.

Examples

- CONVERT (SYMBOL (0) ,″AB″)
  Returns A
- CONVERT (SYMBOL (1) ,″AB″)
  Returns B
- CONVERT (SYMBOL (2) ,″AB″)
  Returns ″none″
- CONVERT (ASCII , ATOETable)
  Converts ASCII text to EBCDIC based on values in a table named **ATOETable**

DATETONUMBER

Use DATETONUMBER to perform arithmetic on dates.

The DATETONUMBER function returns an integer that results from counting the number of days from December 31, 1864, to the specified date.

Syntax:

DATETONUMBER (single-date-expression)

Meaning:

DATETONUMBER (date_to_convert)

Returns:
A single integer

The DATETONUMBER function converts a date to an integer. The resulting integer represents the number of days since December 31, 1864, where using DATETONUMBER with a date of January 1, 1865 returns the integer value 1.

If the input argument is in error, the function returns the value ″none″.

If the date format specified by the input argument does not include century, the century is determined based on the **Century** map setting using the **CCLookup** parameter or the current century.

Examples

- DaysBetween = DATETONUMBER (StopDate) - DATETONUMBER (StartDate)

  This expression could be used in a map rule to produce a value for **DaysBetween**. It converts the **StopDate** and the **StartDate** to integers, then subtracts the resulting two integers, and returns the result as **DaysBetween**.

Related functions

- ADDDAYS
- NUMBERTODATE

DATETOTEXT

The DATETOTEXT function converts a date object or expression to a text item.

Syntax:
DATETOTEXT (single-date-expression)

Meaning:
DATETOTEXT (date_to_convert)

Returns:
A single text item

If *date_to_convert* is a date object name, this returns the date as a text item formatted according to the presentation of the date object.

If *date_to_convert* is a date expression produced by a function, this returns the date as a text item formatted according to the presentation of the output argument of that function.

Examples

- DATETOTEXT (ShipDate)

  In this example, **ShipDate** is converted from a date to text. If **ShipDate** has a CCYYMMDD presentation, the resulting text item will have that presentation, as well.

- DATETOTEXT (CURRENTDATETIME (″{MM/DD/CCYY}″))

  In this example, CURRENTDATETIME evaluates and returns a date in MM/DD/CCYY format. Then DATETOTEXT evaluates and returns a text string that is that date in MM/DD/CCYY format.

For example, use DATETOTEXT, to do text concatenation. The FROMDATETIME function provides greater flexibility in specifying the format of the resulting text item.

Related Functions

- FROMDATETIME
- NUMBERTOTEXT
- TEXT
- TEXTTODATE
- TEXTTONUMBER
- TEXTTOTIME
- TIMETOTEXT
- TODATETIME

FROMBASETEN

You can use FROMBASETEN when you need to convert numbers to a base other than 10.

The FROMBASETEN function converts an integer to a text item that can be interpreted as a number, using positional notation of the base specified.

Syntax:

FROMBASETEN (single-integer-expression , single-integer-expression)

Meaning:

FROMBASETEN (positive_integer_to_convert , base_to_convert_to)

Returns:

A single text item

FROMBASETEN returns a text item that results from converting *positive_integer_to_convert* to a text item that can be interpreted as a number using positional notation of the base specified by *base_to_convert_to*.

If *base_to_convert_to* is less than 2 or greater than 36, FROMBASETEN evaluates to "none". Resulting text item characters A-Z are interpreted as digits having decimal values from 10-35, respectively. The characters returned are uppercase.

Example

- FROMBASETEN (18 , 2)

  Returns the value 10010
- FROMBASETEN (123 , 8)

  Returns the value 173

Related function

- TOBASETEN

FROMDATETIME

The FROMDATETIME function converts a date/time item to a text string of a specified format. For example, you can use FROMDATETIME to convert a date-time item into a string for parsing or concatenation. You can also use this function to access the individual parts of a date or time, such as the month, day, year, and so forth.

Syntax:

FROMDATETIME (single-character-date-time-item
[, single-text-expression])

Meaning:

FROMDATETIME (date_time [, date_time_format_string])

Returns:

A single character text item

FROMDATETIME returns the date/time specified by *date_time* as a text item with the format specified by *date_time_format_string*. If *date_time_format_string* is not specified, *date_time* will be returned in the same format as the *date_time*.

The *date_time_format_string* must conform to the date/time format strings as described in the ″Format strings″ section.

Examples

- HeaderLine=″Today is″ + FROMDATETIME (CURRENTDATE (),″{MON DD, CCYY}″)

  If the current system date is March 3, 1999, this rule evaluates to Today is Mar 03, 1999.
- FROMDATETIME (TransactionTimeStamp Column:.:TransHistory)

  If the value of **TransactionTimeStamp Column** is 10:14 A.M. on February 3, 2000 and its format is defined as ″{CCYYMMDDHHMM}″, 200002031014.
- EXTRACT (Expense:Report, FROMDATETIME (Date:Expense:Report, ″{MM}″) = ″03″)

  In this example, the FROMDATETIME function is used in the condition expression of the EXTRACT function to identify all expenses in the month of March.

Related functions

- CURRENTDATE
- CURRENTDATETIME
- CURRENTTIME
- DATETOTEXT
- TODATETIME

FROMNUMBER

The FROMNUMBER function converts a number to a text string of a specified format.

Use FROMNUMBER when you need to convert a number item into a string for parsing or concatenation.

Syntax:

FROMNUMBER (single-character-number-item [, single-text-expression])

Meaning:
FROMNUMBER (number_to_convert [, number_format_string])

Returns:
A single character text item

FROMNUMBER returns the number specified by *number_to_convert* as a text item with the format specified by *number_format_string*. If *number_format_string* is not specified, *number_to_convert* will be returned in the same format as the *number_to_convert*.

The *date_time_format_string* must conform to the number format strings as described in the ″Format strings″ section.

Examples

- Greeting = ″You are caller number ″ + FROMNUMBER (SeqNo:.:History, ″{#’,’###}″) + ″!″

  If the value of **SeqNo** is 2348192, this rule evaluates to ″You are caller number 2,348,192!″
- ″$″ + FROMNUMBER(CurrentRate Field:.:Schedule, ″{#’,’###’.’2##2}″)

  If the value of **CurrentRate Field** is 15.875, ″$15.88″.

Related functions

- DATETONUMBER
- NUMBERTODATE
- NUMBERTOTEXT
- TONUMBER

HEXTEXTTOSTREAM

HEXTEXTTOSTREAM is the reverse of STREAMTOHEXTEXT. You can use the HEXTEXTTOSTREAM function to assign a binary text value to a character text item represented by hexadecimal pairs.

HEXTEXTTOSTREAM returns a binary text stream whose value is the evaluation of input character text represented by hexadecimal pairs.

Syntax:
HEXTEXTTOSTREAM (single-text-expression)

Meaning:
HEXTEXTTOSTREAM (series_of_hex_pairs)

Returns:
A single byte stream item

This function returns a binary text stream item whose value is the evaluation of input character text in *series_of_hex_pairs*, ignoring <WSP> characters between the hexadecimal pairs. White space characters include space, horizontal tab, carriage return, and line feed characters.

Input formats

The following table shows an example of input in its character text representation as viewed through the character editor, and in its ASCII code representation (binary text stream) as viewed through the hex editor. Each pair of binary text in the hex view represents one character in the character view of the character text.

Input (″41 42 43 44″)	**Editor View**	**Value**
Character text (hex pairs)	Character	″41 42 43 44″
ASCII code representation (binary text stream)	Hex	0x343120343220343320343434

Examples

- HEXTEXTTOSTREAM (″41 42 43 44″)

  Returns the evaluated value of the input (ASCII) character text string ″41 42 43 44″ as the output (ASCII) character text string ″ABCD″ as viewed in the character editor. (The hex view of the input is 0x343120343220343320343434. The hex view of the output is 0x41424344.)
- HEXTEXTTOSTREAM (″0D 0A 00″)

  Returns the evaluated value of the input (ASCII) character text string ″0D 0A 00″ as the output (ASCII) character text string ″<CR><LF><NULL>″ as viewed in the character editor. (The hex view of the input is 0x3044203041203030. The hex view of the output is 0x0D0A00.)

  See Design Studio Introduction documentation for a list of special symbols.

Related functions

- SYMBOL
- STREAMTOHEXTEXT

INT

You can use the INT function when you need only the integer portion of a number.

Syntax:

INT (single-number-expression)

Meaning:

INT (number_to_convert)

Returns:

A single integer

INT returns the integer portion of a number. The result is the integer part of *number_to_convert*. Any fractional part after the decimal point is dropped.

Examples

- INT (1.45)

  Returns 1

- INT (3.6)

  Returns 3
- INT (Purchase:Amt - Discount:Amt)

  Subtracts **Discount:Amt** from **Purchase:Amt** and returns the result as a whole number.

Related functions

- MOD
- ROUND
- TRUNCATE

NUMBERTODATE

You can use the NUMBERTODATE function to perform a calculation on a date.

NUMBERTODATE converts an integer to a date, where the integer is the number of days from December 31, 1864, to the specified date. Only positive, non-zero values can be specified as valid parameters.

.

Syntax:

NUMBERTODATE (single-integer-expression)

Meaning:

NUMBERTODATE (integer_to_convert)

Returns:

A single date

After converting an integer to a date, if the result is being assigned to a date object, the resulting date is in the presentation for that output. If the resulting date is not being assigned to an object, it has a CCYYMMDD presentation.

When the year exceeds four digits, the output will display hash characters (#) for the CCYY values because the field is defined to be only four digits in length.

Examples

- NUMBERTODATE (StartDate)

  This example converts the **StartDate** value from an integer to a date that is in CCYYMMDD format.

Related functions

- ADDDAYS
- DATETONUMBER
- FROMDATETIME
- TODATETIME

NUMBERTOTEXT

The NUMBERTOTEXT function converts a character number to a text item that looks like the original object.

You can use NUMBERTOTEXT when you need an object that is defined as a number converted to an object defined as text. This is useful when you need to concatenate text, however, the FROMNUMBER function provides greater flexibility in specifying the format of the resulting text item.

Syntax:
NUMBERTOTEXT (single-number-expression)

Meaning:
NUMBERTOTEXT (number_to_convert)

Returns:
A single text item

The resulting text looks like the input argument. The result is truncated, if necessary.

Examples

- NUMBERTOTEXT (ROUND (1000 - 24.75, 3))

  This example converts the result of the calculation (rounded to 3 decimal places) to text, resulting in 975.250.
- NUMBERTOTEXT (PurchaseNumber)

  This example converts **PurchaseNumber** from a number to text.

Related functions

- FROMNUMBER
- TEXTTONUMBER
- TODATETIME
- TONUMBER

PACK

The PACK function converts an integer to a text item that can be interpreted as a packed decimal number.

The sign values for packed data are as follows:

- C for positive (+)
- D for negative (-)
- F for unsigned, which is read as positive

Syntax:
PACK (single-integer-expression)

Meaning:
PACK (integer_to_conver)

Returns:
A single text item

In a packed decimal number, each half-byte is a digit, except for the last half-byte of the rightmost byte, which contains a sign.

Examples

- PACK (314)

  Returns ″1L″ and results in the hex value 31 4C (which, in ASCII, looks like 1L)

- PACK ((Unit Price * Quantity) * 100)

  In this example, the packed number has two implied decimal places. Because PACK does not accept decimal places, including implied ones, the nested arithmetic expression, **Unit Price * Quantity**, is multiplied by 100 before rounding.

  Define items as having a packed decimal number presentation. Then, when mapping to or from these items, the conversion to and from packed decimal is automatically performed as needed.

Related function

- UNPACK

PACKAGE

The PACKAGE function converts a group or item object to a text item, including its initiator, terminator, and any delimiters it contains.

Syntax:

PACKAGE (single-object-expression)

Meaning:

PACKAGE (object_to_convert)

Returns:

A single text item

The PACKAGE function converts *object_to_convert*, which must be a type reference to a text item, including the type reference's initiator, terminator, and all delimiters. PACKAGE differs from TEXT in that it includes the initiator and terminator of the specified type reference.

Examples

- PACKAGE (Record:Card)

  Returns: #1339X10A,491.38,Green,42x54@

  For this example, the group **Record** has an initiator of "**#**", a terminator of "**@**" and a delimiter of "**,**". The data looks like this: "#1339X10A,491.38,Green,42x54@".

Related functions

- DATETOTEXT
- NUMBERTOTEXT
- SERIESTOTEXT
- TIMETOTEXT
- TEXT

  PACKAGE differs from TEXT because it includes the initiator and terminator of the input object.

QUOTEDTOTEXT

The QUOTEDTOTEXT decoding function converts a single (text or binary) data object in Quoted-Printable format to a single data object that represents the original text or binary object.

Syntax:

QUOTEDTOTEXT (single-object-expression)

Meaning:
QUOTEDTOTEXT (quoted_printable_object_to_convert)

Returns:
A single-text-item

You can use this function to decode data that was previously encoded to RFC 1767 Quoted-Printable encoding. Any hexadecimal representations are decoded to the appropriate ASCII character and any soft breaks are removed.

If an error occurs during conversion, no data is returned.

Example

QUOTEDTOTEXT(″Unencoded data=0CEncoded=″)

Output: Unencoded data<FF>Encoded

SERIESTOTEXT

You can use the SERIESTOTEXT function to project your input data as a series and to interpret it as a text item for output.

SERIESTOTEXT converts a contiguous or non-contiguous series to a text item.

Syntax:
SERIESTOTEXT (series-object-expression)

Meaning:
SERIESTOTEXT (series_to_convert)

Returns:
A single text item

SERIESTOTEXT returns a text item containing the concatenation of the series of the input argument, including nested delimiters but excluding initiators and terminators.

Examples

In this example, you have the following data that represents bowler information for a bowling league:

Andrews, Jessica:980206:JBC:145:138:177:159

Little, Randy:980116:BBK:175:168

Wayne, Richard:980102:JBC:185:204:179:164:212

Each record consists of the bowler's name, the date of their last game played, a team code and one or more bowling scores. **Record** is defined as a group that is infix-delimited by a colon.

Using the rule:

= SERIESTOTEXT (Score Field:Bowler:Input)

the following results are produced, which is the concatenation of all of the scores for all of the bowlers, even though the scores are not all contiguous within the data:

145138177159175168185204179164212

However, if the rule was changed, for instance, to concatenate the list of scores to the bowler's name:

= BowlerName Field:Bowler:Input + " ->" +
SERIESTOTEXT (Score Field:Bowler:Input)

the following output would be produced:

Andrews, Jessica -> 145138177159

Little, Randy -> 175168

Wayne, Richard -> 185204179164212

In this example, you have an input number that is of variable size, followed by a name. There is no syntax that separates the number from the name. You can define the number as a group with **Byte(s)** as a component and provide a component rule for **Byte(s)**, such as:

ISNUMBER ($)

Based on this, the number can be distinguished from the name. When mapping, collect all the bytes of the number back again. You can use SERIESTOTEXT to do this.

Related functions

- PACKAGE
- TEXT

STREAMTOHEXTEXT

Use STREAMTOHEXTEXT to assign a character text value to a binary text stream item.

The STREAMTOHEXTEXT function returns a character text string represented by hex pairs whose value is the evaluation of an input binary text stream.

This function is the reverse of the HEXTEXTTOSTREAM function.

Syntax:
STREAMTOHEXTEXT (single-byte-stream-item)

Meaning:
STREAMTOHEXTEXT (single-byte-stream-item)

Returns:
A series of hex pairs

STREAMTOHEXTEXT returns a string of hex pairs whose value is the evaluation of input binary text in *series_of_hex_pairs*.

Input formats

The following table shows an example of input in its ASCII code representation (binary text stream) as viewed through the hex editor, and in its character text representation as viewed through the character editor. Each four-character grouping of binary text in the hex view represents one character in the character view.

Input (0x41424344)	Editor View	Value
ASCII code representation (binary text stream)	Hex	0x41424344
	Character	″ABCD″

Examples

- STREAMTOHEXTEXT (0x41424344)

  Returns the evaluated value of the input (ASCII) binary text stream **0x41424344** as the output (ASCII) binary text stream 0x3431343234333434 as viewed in the hex editor. (The character view of the input is ″ABCD″. The character view of the output is ″41424344″.)
- STREAMTOHEXTEXT (0x0D0A00)

  Returns the evaluated value of the input binary text stream **0x0D0A00** as the output 0x304430413030 as viewed in the hex editor. (The character view of the input is ″<CR><LF><NULL>″. The character view of the output is ″0D0A00″.)

Related functions

- HEXTEXTTOSTREAM
- SYMBOL

SYMBOL

You can use SYMBOL to include a symbol in your output.

The SYMBOL function returns a one-byte character that is the ASCII character equivalent of a specified decimal value.

Syntax:

SYMBOL (single-integer-expression)

Meaning:

SYMBOL (decimal_value)

Returns:

A single one-byte text item

The value of the resulting item is the ASCII character equivalent of *decimal_value*. Valid input values are 0 to 255. Values outside of this range return ″none″.

A listing of decimal values (0-127) and their ASCII character equivalents is included in the Design Studio Introduction documentation.

Examples

- SYMBOL (13)

  Produces a carriage return
- SYMBOL (13) + SYMBOL (10)

Produces a carriage return/linefeed

You can accomplish the same result using the angle-brackets around the decimal value. For example, the second example above would be equivalent to <CR><LF> or <<0D>><<0A>>

Related functions

- HEXTEXTTOSTREAM
- STREAMTOHEXTEXT

TEXTTOBASE64

The TEXTTOBASE64 encoding function converts a text item to Base64 format.

Syntax:
TEXTTOBASE64 (single-object-expression)

Meaning:
TEXTTOBASE64 (text_object_to_convert)

Returns:
A single-text-item

The TEXTTOBASE64 function receives a text object, converts the text to Base64 format, and returns a single data object that represents the Base64 encoding of the original text object. If an error occurs during conversion, no data is returned.

Example

TEXTTOBASE64(″Some Example Data″)

Output: U29tZSBFeGFtcGxlIERhdGE=

TEXTTOBCD

The TEXTTOBCD function converts a text item from decimal digits to BCD (Binary Coded Decimal) format.

Syntax:
TEXTTOBCD (single-integer-text-expression)

Meaning:
TEXTTOBCD (text_to_be_converted)

Returns:
A single BCD-formatted text item

TEXTTOBCD converts *text_to_be_converted* (which consists of decimal digits) to BCD format. In this format, each byte contains two decimal digits represented as binary numbers. If there is an odd number of decimal digits in the input, the high-order half-byte of the leftmost output byte will contain the decimal value 15 (hex ″F″).

If anything other than a decimal digit is encountered in the input, TEXTTOBCD returns ″none″.

Examples

- TEXTTOBCD (″1234″)

  Returns the hexadecimal value x'1234'
- TEXTTOBCD (″123A″)

  Returns ″none″
- TEXTTOBCD (″123″)

  Returns the hexadecimal value x'F123'

  In this example, the values shown as input (″123″) are meant to represent character items in the native character set to the machine on which the map is running. On a personal computer, ″123″ would contain the ASCII characters for the digits that have the hexadecimal values ″31″, ″32″, and ″33″. The output, described as ″the hexadecimal value `F123'″, consists of the two binary bytes ″F1″ and ″23″.

  On an IBM mainframe the input string would contain EBCDIC characters for the digits that have the hexadecimal values ″F1″, ″F2″, ″F3″, Âº, but the output would be the same as the personal computer output.

Related functions

- BCDTOHEX
- BCDTOINT
- BCDTOTEXT

TEXTTODATE

The TEXTTODATE function converts text item from CCYYMMDD or YYMMDD format to a date.

You can use the TEXTTODATE to convert a text item to a date, however, the TODATETIME function provides greater flexibility for specifying the resulting date-time format .

Syntax:

TEXTTODATE (single-text-expression)

Meaning:

TEXTTODATE (text_to_convert_to_date)

Returns:

A single date

The *text_to_convert_to_date* must be in either CCYYMMDD or YYMMDD format. If the *text_to_convert_to_date* is in error (for example, it is not a valid date), the result is ″none″.

If *text_to_convert_to_date* is in YYMMDD format and is being assigned to a date format that includes a century, the century is determined based on the century run option setting using the **CCLookup** parameter or the current century.

Examples

- TEXTTODATE (″990114″)

  Returns a single date item with the value of January 14 in the year 99
- TEXTTODATE (OrderDate)

  Returns **OrderDate** as a single date item.

Related functions

• DATETOTEXT	• TEXTTOTIME
• NUMBERTOTEXT	• TIMETOTEXT
• TEXTTONUMBER	• TODATETIME

TEXTTONUMBER

Use TEXTTONUMBER to convert text to a number.

The TONUMBER function provides greater flexibility for specifying the format of the text item that is to be converted to a number.

Syntax:

TEXTTONUMBER (single-text-expression)

Meaning:

TEXTTONUMBER (text_to_convert_to_number)

Returns:

A single character number

The *text_to_convert_to_number* must be in integer or ANSI-formatted (floating point) presentation. The resulting number looks like the input argument, however, nonsignificant zeroes to the right of the decimal separator will be truncated. If the input argument is in error (for example, it is not a recognizable as a valid number), the result is ″none″.

When specified as in ANSI-formatted presentation, the text string must meet the following requirements:

- The decimal point can be a period, a comma, or ″none″.
- The leading sign can be a plus sign, a minus sign, or ″none″.
- No thousands separator is allowed.

Examples

- TEXTTONUMBER (OrderQty)

  Returns **OrderQty** as a character number item

Related functions

- DATETOTEXT
- FROMNUMBER
- NUMBERTOTEXT
- TEXTTODATE
- TEXTTOTIME
- TIMETOTEXT

TEXTTOQUOTED

The TEXTTOQUOTED encoding function converts a text or binary item to Quoted-Printable format.

Syntax:
TEXTTOQUOTED (single_object_expression)

Meaning:
TEXTTOQUOTED (object_to_convert)

Returns:
A single text or binary item

You can use this function to encode data that largely consists of octets that correspond to printable characters in the US-ASCII character set. It encodes the data in such a way that the resulting octets are unlikely to be modified by mail transport.

Quoted-Printable lines cannot be longer than 76 characters. Data with lines greater than 76 characters are broken up and indicated with soft breaks of ″=<CR><LF>″.

If an error occurs during conversion, no data is returned.

Example

This example converts the <FF> (form feed) character, into a hexadecimal representation, and indicates the end of data with a soft break.

TEXTTOQUOTED(″Unencoded data<FF>Encoded″)

Output: Unencoded data=0CEncoded=

The data is converted using the Quoted-Printable encoding as per RFC 1767.

TEXTTOTIME

Use TEXTTOTIME when you want to convert an object defined as text that is in HHMM or HHMMSS presentation, to an item defined as time. For greater flexibility, use the TODATETIME function for specifying the format of the text item that is to be converted to a date/time.

Syntax:
TEXTTOTIME (single-text-expression)

Meaning:
TEXTTOTIME (text_to_convert_to_time)

Returns:
A single time

The *text_to_convert_to_time* must be in HHMM or HHMMSS presentation. HH is a two-digit hour in a 24-hour format. If the result is being assigned to a time object, the resulting time looks like the output object. Otherwise, the resulting time looks like the input argument. If the input argument is in error (for example, it is not a valid time), the result is ″none″.

Examples

- TEXTTOTIME (CallTime)

  Returns **CallTime** as a time item

Related functions

• DATETOTEXT	• TEXTTONUMBER
• FROMDATETIME	• TIMETOTEXT
• NUMBERTOTEXT	• TODATETIME
• TEXTTODATE	

TEXTTOTIME

Use TEXTTOTIME when you want to convert an object defined as text that is in HHMM or HHMMSS presentation, to an item defined as time. For greater flexibility, use the TODATETIME function for specifying the format of the text item that is to be converted to a date/time.

Syntax:

TEXTTOTIME (single-text-expression)

Meaning:

TEXTTOTIME (text_to_convert_to_time)

Returns:

A single time

The *text_to_convert_to_time* must be in HHMM or HHMMSS presentation. HH is a two-digit hour in a 24-hour format. If the result is being assigned to a time object, the resulting time looks like the output object. Otherwise, the resulting time looks like the input argument. If the input argument is in error (for example, it is not a valid time), the result is ″none″.

Examples

- TEXTTOTIME (CallTime)

  Returns **CallTime** as a time item

Related functions

• DATETOTEXT	• TEXTTONUMBER
• FROMDATETIME	• TIMETOTEXT
• NUMBERTOTEXT	• TODATETIME
• TEXTTODATE	

TOBASETEN

The TOBASETEN function converts a text item that can be interpreted as a number, using positional notation of the base specified, to a base 10 number.

Syntax:

TOBASETEN (single-text-expression , single-integer-expression)

Meaning:

TOBASETEN (text_to_convert , base_to_convert_from)

Returns:

A single integer

TOBASETEN returns a number that results from converting *text_to_convert* that can be interpreted as a number, using positional notation of the base specified by *base_to_convert_from*, to its base 10 representation. Text item characters A-Z are interpreted as digits having decimal values from 10-35, respectively.

If *base_to_convert_from* is less than 2 or greater than 36, TOBASETEN evaluates to ″none″. If *text_to_convert* contains a character that is not alphanumeric or is not in the range specified by *base_to_convert_from*, TOBASETEN returns ″none″.

Examples

- TOBASETEN (″A″ , 16)

  Returns the value 10
- TOBASETEN (″10″ , 36)

  Returns the value 36
- TOBASETEN (″A0″ , 15)

  Returns the value 150
- TOBASETEN (″A0″ , 5)

  Returns the value ″none″

Related functions

- FROMBASETEN

TODATETIME

The TODATETIME function converts a text string of a specified format to a date-time item.

Syntax:

TODATETIME (single-character-text-expression
[, single-text-expression])

Meaning:

TODATETIME (text_to_convert [, date_time_format_string])

Returns:

A single character date item

TODATETIME returns the date-time that corresponds to the value specified by *text_to_convert*, which is in the format specified by *date_time_format_string*. If *date_time_format_string* is not specified, it will be assumed that *text_to_convert is* in [CCYYMMDDHH24MMSS] format.

The *date_time_format_string* must conform to the date-time format strings as described in ″Format strings″.

Examples

- TODATETIME (″05/14/1999@10:14pm″ , ″{MM/DD/CCYY}@{HH12:MMAM/PM}″)

  In this example, a text string containing a date and time is converted to a date-time item.

- RptDate = TODATETIME (RIGHT (GETRESOURCENAME(), 8) , ″CCYYMMDD″)

  Assume that you receive a file that contains historical data. The name of the file identifies the date of the historical data. For example, a filename of **19960424** indicates that the data was produced on April 24, 1996. To map this date to **RptDate**, the TODATETIME function could be used with the RIGHT and GETRESOURCENAME functions.

Related functions

- CURRENTDATE
- CURRENTDATETIME
- CURRENTTIME
- TEXTTODATE
- TEXTTOTIME

TONUMBER

The TONUMBER function converts a text string of a specified format to a number.

Syntax:

TONUMBER (single-character-text-expression [, single-text-expression])

Meaning:
TONUMBER (text_to_convert [, number_format_string])

Returns:
A single character number item

TONUMBER returns the number that corresponds to the value specified by *text_to_convert*, which is in the format specified by *number_format_string*. If *number_format_string* is not specified, it will be assumed that *text_to_convert* is in ANSI decimal format (for example, ″{L-####[’.’##]}″).

The *number_format_string* must conform to the number format strings as described in the ″Format strings″ section.

Examples

- TONUMBER(*text_to_convert*, ″{L+’$’#’,’###}″)

  L+’$’ indicates the leading dollar sign is positive. That leading sign and the comma separators are removed when the text is converted to a number.

  Input String: $123,000,000

  Output: 123000000

- TONUMBER(*text_to_convert*, ″{####T-}″)

  Four number signs are required for each whole number, regardless of the actual number of digits in the number.

Input string:	Output:	Note:
12345-	-12345	The output becomes a negative number.
67890	67890	No change occurs.
345-	-345	The output becomes a negative number.

- TONUMBER(*text_to_convert*, ″{####T+’K’-}″)

If an invalid character, such as an X, is encountered, nothing is returned.

If a **K** is encountered, it is treated as a positive indicator.

Input string:	Output:	Note:
11212-	-11212	The output becomes a negative number.
67890X		The X is an invalid character. No number is returned.
54354	54354	No change occurs.
34567K	34567	The K is recognized as a positive sign. The character is removed and the number is returned as a positive.
345-	-345	The output becomes a negative number.

- TONUMBER(*text_to_convert*, "{L-'('#','###T-')'}")

  The parentheses indicating a negative number are removed and replaced with a negative sign.

  Comma separators are removed when the text is converted to a number.

Input string:	Output:	Note:
(12,345)	-12345	The output becomes a negative number. The comma separator is removed.
67,890	67890	The comma separator is removed.
(345)	-345	The output becomes a negative number.

- TONUMBER(*text_to_convert*, "{#[',']###['.'##5]T+'K'-}")

  The optional comma separators are removed, but the decimal points and decimal values are retained.

Input string:	Output:	Note:
54,345.098	54354.098	The comma separator is removed.
67890.0X		The X is an invalid character. No number is returned.
11213-	-11213	The output becomes a negative number.
34567K	34567	The K is recognized as a positive sign. The character is removed and the number is returned as a positive.
345.1-	-345.1	The output becomes a negative number.

Related functions

- DATETONUMBER
- FROMNUMBER
- NUMBERTODATE
- NUMBERTOTEXT

UNPACK

You can use the UNPACK function to do arithmetic with a packed decimal number or to move a packed decimal value into a numeric item.

UNPACK converts text that can be interpreted as a packed decimal number to a signed integer item.

The sign values for packed data are as follows:

- C for positive (+)
- D for negative (-)
- F for unsigned, which is read as positive

Syntax:
UNPACK (single-fixed-size-text-expression)

Meaning:
UNPACK (text_to_unpack)

Returns:
A single signed integer

UNPACK returns a signed integer representing the value *text_to_unpack,* which is a packed decimal number. If the *text_to_unpack* cannot be interpreted as a valid packed decimal, UNPACK evaluates to ″none″.

In a packed decimal number, each half-byte is a digit, except for the last half-byte of the rightmost byte, which contains a sign.

Examples

- UNPACK (″1L″) returns 314

  The ASCII string ″1L″ in hex is 31 4C, which, when interpreted as a packed number, results in (positive) 314. This example returns the value ″+314″.

  The hexadecimal representation of the value ″1L″ is x`14C', where C in the rightmost half-byte represents a positive sign.
- UNPACK (UnitPrice) / 100 * QuantityOrdered

  UnitPrice is unpacked and divided by 100 (to convert it from an integer to a number with two decimal places) and then multiplied by the *QuantityOrdered*.

  You can define items as having a packed decimal number presentation. Then, when mapping to or from these items, the conversion to and from packed decimal is automatically performed as needed.

Related functions

- PACK

UNZONE

You can use the UNZONE function to convert a text item that represents a zoned (signed) number to an integer.

UNZONE converts a text item that can be interpreted as a number with a super-imposed sign in the rightmost byte (called zoned or signed data) to a signed integer item.

Syntax:
UNZONE (single-text-expression)

Meaning:
UNZONE (text_to_unzone)

Returns:
A single integer

UNZONE returns a signed integer representing the value *text_to_unzone* that is an integer in zoned (signed) format. If the *text_to_unzone* cannot be interpreted as a valid zoned number, UNZONE evaluates to ″none″.

Zoned integers have a series of digits except for the rightmost byte. The rightmost byte is a digit with a super-imposed sign. See ″Positive zoned values″ for a list of rightmost byte values.

Examples

- UNZONE (″123D″)

  Returns 1234
- UNZONE (″1234″)

  Returns 1234
- UNZONE (TaxRate) / 1000 * Income

  TaxRate is converted from zoned format to a signed decimal and divided by 1000 (to convert it to a number with three decimal places), and then multiplied by **Income**.

  You can define items as having a zoned character number presentation. Then, when mapping to or from these items, the conversion to and from zoned decimal is automatically performed as needed.

Related functions

- ZONE

ZONE

Use ZONE to convert a number to a zoned (signed) number.

The ZONE function converts a signed integer item to a text item that can be interpreted as a number with a superimposed sign in the rightmost byte (called zoned or signed).

You can define items as having a zoned character number presentation. Then, when mapping to or from these items, the conversion to and from zoned decimal is performed automatically, as needed.

Syntax:
ZONE (single-integer-expression , single-integer-expression)

Meaning:
ZONE (integer_to_convert , sign_indicator)

Returns:
A single text item

ZONE returns a text string that represents a zoned (signed) number representing *integer_to_convert*.

Zoned integers have a sequence of digits except for the rightmost byte. The rightmost byte is a digit with a super-imposed sign. The *sign_indicator* specifies

whether a super-imposed sign is required for positive integers, where 0 specifies no sign is required and any other value specifies that a sign is required for positive integers.

The following tables show positive and negative values of the numbers 1230 to 1239, with a sign indicator of 0 (no sign is required for positive values) or a sign of 1 (include a sign for both positive and negative values) in the rightmost byte.

Table 1. Positive Zoned Values

Integer Value	sign_indicator = 0	sign_indicator = 1
1230	1230	123{
1231	1231	123A
1232	1232	123B
1233	1233	123C
1234	1234	123D
1235	1235	123E
1236	1236	123F
1237	1237	123G
1238	1238	123H
1239	1239	123I

Table 2. Negative Zoned Values

Integer Value	sign_indicator = 0	sign_indicator = 1
-1230	123}	123}
-1231	123J	123J
-1232	123K	123K
-1233	123L	123L
-1234	123M	123M
-1235	123N	123N
-1236	123O	123O
-1237	123P	123P
-1238	123Q	123Q
-1239	123R	123R

Examples

- ZONE (1234 , 0)

  Returns 1234
- ZONE (1234 , 1)

  Returns 123D
- ZONE (-1234 , 1)

  Returns 123M
- ZONE (INT (UnitPrice * 100) , 1)

  UnitPrice is multiplied by 100 to move the first two decimal places to the left of the decimal sign. The result of this calculation is converted to an integer using the INT function. Finally, the result of the INT is converted to zoned format, using a sign for positive values.

Related function

- UNZONE

Chapter 8. Date/time functions

ADDDAYS

The ADDDAYS function adds a specified number of days to a given date.

You cannot use ADDDAYS on a date/time object that does not specify the day of the month because an error will occur.

Syntax:

ADDDAYS (single-date-expression, single-integer-expression)

Meaning:

ADDDAYS (any_date, number_of_days_to_add)

Returns:

A single date

The ADDDAYS function returns the date, which results from adding *number_of_days_to_add* to *any_date*. If *any_date* has a presentation that does not contain a century, the century is determined based on the Century > CCLookup map setting (when the Century > Switch = ON), or the current century (when the Century > Switch = OFF).

When the year exceeds four digits, the output will display hash characters (#) for the CCYY values because the field is defined to be only four digits in length.

Examples

- You can use the ADDDAYS function to increment a date by a fixed number of days, such as to calculate a **DueDate** that is always 30 days after the **InvoiceDate**.

  To produce a date in the output, such as a ship date, you might need to add a variable number of days to a date in the input. For example: ADDDAYS (PODate, LeadTime).
- ADDDAYS (InvoiceDate, 10)

  Returns the date, which results from adding 10 days to the value of **InvoiceDate**.
- ADDDAYS (InvoiceDate, DaysTilDue)

  Returns the date that results from adding the value of **DaysTilDue** to the **InvoiceDate** value.
- ADDDAYS (TODATETIME (″000101″), -1)

  Returns 991231
- ADDDAYS (TODATETIME (″20000101″), 1)

  Returns 20000102
- ADDDAYS (TODATETIME (″10/20/1996″, ″{MM/DD/CCYY}″), 5)

  Returns 10/25/1996

  In the examples containing TODATETIME, the text literal is first converted to a date, and then the specified number of days is added to that date.

Related functions

• ADDHOURS	• TEXTTODATE
• DATETONUMBER	• TODATETIME
• NUMBERTODATE	

ADDHOURS

The ADDHOURS function returns a time value that is the result of adding a specified number of hours to a given time.

Syntax:

ADDHOURS (single-datetime-item-expression, hours-expressed-as-signed-integer-expression)

Meaning:

ADDHOURS (any_datetime, number_of_hours_to_add)

Returns:

A single datetime item

The ADDHOURS function returns a single datetime item that is advanced from the original value of the single datetime item expression by the specified number of hours expressed as integer expression.

Examples

- ADDHOURS(TODATETIME(″Dec 31, 1999 23:59:00″), 2)
  Returns: Jan 1, 2000 01:59:00
- ADDHOURS(TODATETIME(″Dec 31, 1999 23:59:00″), -25)
  Returns: Dec 30, 1999 22:59:00
- ADDHOURS(TODATETIME(″23:59:00″), 2)
  Returns: 01:59:00

If either argument is invalid, the result is ″none″. If the second argument is ″none″, the result is the first argument. If the first argument contains only a time portion, date changes are ignored. If the first argument contains only a date portion, the time portion 00:00:00 will be used in the calculation. If both arguments are ″none″, the result is ″none″.

Related functions

• ADDDAYS	• TEXTTODATE
• DATETONUMBER	• TODATETIME
• NUMBERTODATE	

ADDMINUTES

The ADDMINUTES function returns a time value that is the result of adding a specified number of minutes to a given time.

Syntax:

ADDMINUTES (single-datetime-item-expression, minutes-expressed-as-signed-integer-expression)

Meaning:

ADDMINUTES (any_datetime, number_of_minutes_to_add)

Returns:

A single datetime item

The ADDMINUTES function returns a single datetime item, advanced from the original value of the single datetime item expression by the specified number of minutes-expressed-as-integer expression.

Examples

- ADDMINUTES(TODATETIME(″Dec 31, 1999 23:59:00″), 2)

  Returns: Jan 1, 2000 00:01:00
- ADDMINUTES(TODATETIME(″Dec 31, 1999 23:59:00″), -25)

  Returns: Dec 31, 1999 23:34:00
- ADDMINUTES(TODATETIME(″23:59:00″), 2)

  Returns: 00:01:00

  If either argument is invalid, the result is ″none″. If the second argument is ″none″, the result is the first argument. If the first argument contains only a time portion, date changes are ignored. If the first argument contains only a date portion, the time portion 00:00:00 will be used in the calculation. If both arguments are ″none″, the result is ″none″.

Related Functions

- DATETONUMBER
- NUMBERTODATE
- TEXTTODATE
- TODATETIME

CURRENTDATE

You can use CURRENTDATE when you need the current date as a transaction processing date, an order received date, or other date that reflects when the data was mapped.

When you need to parse the system date, use CURRENTDATE with the TEXT or FROMDATETIME functions. The function returns the current system date.

Syntax:

CURRENTDATE ( )

Meaning:

CURRENTDATE ( )

Returns:

A single date

CURRENTDATE has no arguments but it does require parentheses.

If being assigned to a date/time output item, the current date is returned in the format specified by that output item. Otherwise, the system date is returned in an MM/DD/YY presentation.

Examples

- StartDate = CURRENTDATE ( )

  In this example, **StartDate** is assigned the value of the current **dateIn** this example, because CURRENTDATE is assigned to an output, it is automatically converted to the presentation of **StartDate**.

  If CURRENTDATE evaluates to 06/24/37 and **StartDate** has a YYMMDD presentation, the result is 370624.
- FROMDATETIME (CURRENTDATE ( ) , ″DD.MON.CCYY″)

  In this example, the current date is returned in DD.MON.CCYY format. If today's date is January 5, 1999, the date returned would be 05.JAN.1999.

Related functions

- CURRENTDATETIME
- CURRENTTIME
- FROMDATETIME
- DATETOTEXT

CURRENTDATETIME

CURRENTDATETIME returns the current system date and time. You can use this function when you need to map the current system date and time to an item that includes both a date and time portion.

.

Syntax:

CURRENTDATETIME ([single-text-expression])

Meaning:

CURRENTDATETIME ([date_time_format_string])

Returns:

A single date-time

CURRENTDATETIME has no arguments but it does require parentheses.

The CURRENTDATETIME function returns the system date and time in the format specified by *date_time_format_string* or with a CCYYMMDDHHMMSS presentation if no *date_time_format_string* is provided.

The *date_time_format_string* must conform to the date/time format strings as described in ″Format strings″.

Examples

- StartDateTime = CURRENTDATETIME ( )

  In this example, **StartDateTime** is assigned the value of the current date and time. Because CURRENTDATETIME is assigned to an output, it is automatically converted to the presentation of **StartDateTime**.

  If CURRENTDATETIME evaluates to 3:04pm on 6/24/1999 and **StartDateTime** has a YYMMDDHH12MM presentation, the result is 9906240304.

- CURRENTDATETIME ("{MM.DD.CCYY HH24:MM}")

  In this example, the current date is returned in MM.DD.CCYY HH24:MM format. If it is currently 4:12 pm on January 5, 1999, the date returned would be 01.05.1999 16:12.

Related Functions

- CURRENTDATE
- CURRENTTIME
- FROMDATETIME
- DATETOTEXT

CURRENTTIME

The CURRENTTIME function returns the current system time.

You can use CURRENTTIME when you need the system time as a transaction processing time, an order-received time, or a time that reflects when the data was mapped.

Syntax:

CURRENTTIME ( )

Meaning:

CURRENTTIME ( )

Returns:

A single time

The CURRENTTIME function returns the system time. If assigned to a date-time output item, the current time is returned in the format specified by that output item. Otherwise, the system time is returned in HH:MM:SS presentation.

Note: CURRENTTIME has no arguments but it does require parentheses.

Examples

- End Time = CURRENTTIME ( )

  In this example, **End Time** is assigned the current time. Because CURRENTTIME is assigned to an output, it is automatically converted to the presentation of **End Time**.

  If CURRENTTIME evaluates to 10:15:02 and **End Time** has an HH12:MM presentation, the result is 1015.
- FROMDATETIME (CURRENTTIME ( ) , "{HH24MMSS}")

  In this example, the current time is returned in HH24MMSS format. If the current time is 10:15:02 pm, the result is 221502.

Related functions

- CURRENTDATETIME
- FROMDATETIME
- DATETOTEXT

DATETONUMBER

Use DATETONUMBER to perform arithmetic on dates.

The DATETONUMBER function returns an integer that results from counting the number of days from December 31, 1864, to the specified date.

Syntax:
DATETONUMBER (single-date-expression)

Meaning:
DATETONUMBER (date_to_convert)

Returns:
A single integer

The DATETONUMBER function converts a date to an integer. The resulting integer represents the number of days since December 31, 1864, where using DATETONUMBER with a date of January 1, 1865 returns the integer value 1.

If the input argument is in error, the function returns the value ″none″.

If the date format specified by the input argument does not include century, the century is determined based on the **Century** map setting using the **CCLookup** parameter or the current century.

Examples

- DaysBetween = DATETONUMBER (StopDate) - DATETONUMBER (StartDate)

  This expression could be used in a map rule to produce a value for **DaysBetween**. It converts the **StopDate** and the **StartDate** to integers, then subtracts the resulting two integers, and returns the result as **DaysBetween**.

Related functions

- ADDDAYS
- NUMBERTODATE

DATETOTEXT

The DATETOTEXT function converts a date object or expression to a text item.

Syntax:
DATETOTEXT (single-date-expression)

Meaning:
DATETOTEXT (date_to_convert)

Returns:
A single text item

If *date_to_convert* is a date object name, this returns the date as a text item formatted according to the presentation of the date object.

If *date_to_convert* is a date expression produced by a function, this returns the date as a text item formatted according to the presentation of the output argument of that function.

Examples

- DATETOTEXT (ShipDate)

In this example, **ShipDate** is converted from a date to text. If **ShipDate** has a CCYYMMDD presentation, the resulting text item will have that presentation, as well.

- DATETOTEXT (CURRENTDATETIME ("{MM/DD/CCYY}"))

  In this example, CURRENTDATETIME evaluates and returns a date in MM/DD/CCYY format. Then DATETOTEXT evaluates and returns a text string that is that date in MM/DD/CCYY format.

  For example, use DATETOTEXT, to do text concatenation. The FROMDATETIME function provides greater flexibility in specifying the format of the resulting text item.

Related Functions

- FROMDATETIME
- NUMBERTOTEXT
- TEXT
- TEXTTODATE
- TEXTTONUMBER
- TEXTTOTIME
- TIMETOTEXT
- TODATETIME

FROMDATETIME

The FROMDATETIME function converts a date/time item to a text string of a specified format. For example, you can use FROMDATETIME to convert a date-time item into a string for parsing or concatenation. You can also use this function to access the individual parts of a date or time, such as the month, day, year, and so forth.

Syntax:

```
FROMDATETIME  (single-character-date-time-item
  [ , single-text-expression ] )
```

Meaning:

FROMDATETIME (date_time [, date_time_format_string])

Returns:

A single character text item

FROMDATETIME returns the date/time specified by *date_time* as a text item with the format specified by *date_time_format_string*. If *date_time_format_string* is not specified, *date_time* will be returned in the same format as the *date_time*.

The *date_time_format_string* must conform to the date/time format strings as described in the "Format strings" section.

Examples

- HeaderLine="Today is" + FROMDATETIME (CURRENTDATE (),"{MON DD, CCYY}")

  If the current system date is March 3, 1999, this rule evaluates to Today is Mar 03, 1999.

- FROMDATETIME (TransactionTimeStamp Column:.:TransHistory)

If the value of **TransactionTimeStamp Column** is 10:14 A.M. on February 3, 2000 and its format is defined as "{CCYYMMDDHHMM}", 200002031014.

- EXTRACT (Expense:Report, FROMDATETIME (Date:Expense:Report, "{MM}") = "03")

  In this example, the FROMDATETIME function is used in the condition expression of the EXTRACT function to identify all expenses in the month of March.

Related functions

- CURRENTDATE
- CURRENTDATETIME
- CURRENTTIME
- DATETOTEXT
- TODATETIME

MAX

The MAX function returns the maximum value from a series of number, date, time, or text values.

Syntax:

MAX (series-item-expression)

Meaning:

MAX (series_of_which_to_find_max)

Returns:

A single number

The result is the maximum value in the input argument series: number, text, or date/time.

Examples

- MAX (UnitPrice:Input)

  If the values for **UnitPrice** are {20, 10, 100}, MAX returns 100.

- MAX(EXTRACT(DueDate:Book:Library, CheckedOut:Book:Library = "Y"))

  Returns the maximum (latest) **DueDate** for a book that is checked out from the library.

Related functions

- MIN

MIN

Use MIN when you need the minimum value from a series of number, date, time, or text values.

The MIN function returns the minimum value from a series.

Syntax:

MIN (series-item-expression)

Meaning:

MIN (series_of_which_to_find_min)

Returns:
A single number

The result is the minimum value of the input series: number, text, or date/time.

Examples

- MIN (UnitPrice:Input)
  If the values for **UnitPrice** are {20,10,100}, MIN returns 10.
- MIN (StartTime:.:Schedule)
  Returns the minimum (earliest) **StartTime** in **Schedule**.

Related functions

- MAX

NUMBERTODATE

You can use the NUMBERTODATE function to perform a calculation on a date.

NUMBERTODATE converts an integer to a date, where the integer is the number of days from December 31, 1864, to the specified date. Only positive, non-zero values can be specified as valid parameters.

.

Syntax:
NUMBERTODATE (single-integer-expression)

Meaning:
NUMBERTODATE (integer_to_convert)

Returns:
A single date

After converting an integer to a date, if the result is being assigned to a date object, the resulting date is in the presentation for that output. If the resulting date is not being assigned to an object, it has a CCYYMMDD presentation.

When the year exceeds four digits, the output will display hash characters (#) for the CCYY values because the field is defined to be only four digits in length.

Examples

- NUMBERTODATE (StartDate)
  This example converts the **StartDate** value from an integer to a date that is in CCYYMMDD format.

Related functions

- ADDDAYS
- DATETONUMBER
- FROMDATETIME
- TODATETIME

TEXTTODATE

The TEXTTODATE function converts text item from CCYYMMDD or YYMMDD format to a date.

You can use the TEXTTODATE to convert a text item to a date, however, the TODATETIME function provides greater flexibility for specifying the resulting date-time format .

Syntax:

TEXTTODATE (single-text-expression)

Meaning:

TEXTTODATE (text_to_convert_to_date)

Returns:

A single date

The *text_to_convert_to_date* must be in either CCYYMMDD or YYMMDD format. If the *text_to_convert_to_date* is in error (for example, it is not a valid date), the result is ″none″.

If *text_to_convert_to_date* is in YYMMDD format and is being assigned to a date format that includes a century, the century is determined based on the century run option setting using the **CCLookup** parameter or the current century.

Examples

- TEXTTODATE (″990114″)

  Returns a single date item with the value of January 14 in the year 99
- TEXTTODATE (OrderDate)

  Returns **OrderDate** as a single date item.

Related functions

• DATETOTEXT	• TEXTTOTIME
• NUMBERTOTEXT	• TIMETOTEXT
• TEXTTONUMBER	• TODATETIME

TEXTTOTIME

Use TEXTTOTIME when you want to convert an object defined as text that is in HHMM or HHMMSS presentation, to an item defined as time. For greater flexibility, use the TODATETIME function for specifying the format of the text item that is to be converted to a date/time.

Syntax:

TEXTTOTIME (single-text-expression)

Meaning:

TEXTTOTIME (text_to_convert_to_time)

Returns:

A single time

The *text_to_convert_to_time* must be in HHMM or HHMMSS presentation. HH is a two-digit hour in a 24-hour format. If the result is being assigned to a time object, the resulting time looks like the output object. Otherwise, the resulting time looks like the input argument. If the input argument is in error (for example, it is not a valid time), the result is ″none″.

Examples

- TEXTTOTIME (CallTime)

  Returns **CallTime** as a time item

Related functions

• DATETOTEXT	• TEXTTONUMBER
• FROMDATETIME	• TIMETOTEXT
• NUMBERTOTEXT	• TODATETIME
• TEXTTODATE	

TIMETOTEXT

You can use the TIMETOTEXT function to perform text concatenation. For greater flexibility, use the FROMDATETIME function for specifying the format of the resulting text item.

TIMETOTEX converts a time object or expression to a text item.

Syntax:

TIMETOTEXT (single-time-expression)

Meaning:

TIMETOTEXT (time_to_convert_to_text)

Returns:

A single text item

If *time_to_convert_to_text* is a time object name, this returns the time as a text item formatted according to the presentation of the input date object.

If *time_to_convert_to_text* is a time expression produced by a function, this returns the time as a text item formatted according to the presentation of the output argument of that function.

Examples

- TIMETOTEXT (LeadTime)

  In this example, **LeadTime** is converted from a time to text. If **LeadTime** has an HH:MM presentation, the resulting text item will be of that presentation.
- TIMETOTEXT (CURRENTDATETIME (″{HH:MM:SS}″))

  Here, CURRENTDATETIME evaluates and returns a time in HH:MM:SS format. Then, TIMETOTEXT evaluates and returns a text string that is that time in HH:MM:SS format.

Related functions

- DATETOTEXT
- FROMDATETIME
- NUMBERTOTEXT
- TEXTTODATE
- TEXTTONUMBER
- TEXTTOTIME
- TODATETIME

TODATETIME

The TODATETIME function converts a text string of a specified format to a date-time item.

Syntax:

```
TODATETIME ( single-character-text-expression
 [ , single-text-expression ] )
```

Meaning:

TODATETIME (text_to_convert [, date_time_format_string])

Returns:

A single character date item

TODATETIME returns the date-time that corresponds to the value specified by *text_to_convert*, which is in the format specified by *date_time_format_string*. If *date_time_format_string* is not specified, it will be assumed that *text_to_convert is* in [CCYYMMDDHH24MMSS] format.

The *date_time_format_string* must conform to the date-time format strings as described in ″Format strings″.

Examples

- TODATETIME (″05/14/1999@10:14pm″ , ″{MM/DD/CCYY}@{HH12:MMAM/PM}″)

  In this example, a text string containing a date and time is converted to a date-time item.

- RptDate = TODATETIME (RIGHT (GETRESOURCENAME(), 8) , ″CCYYMMDD″)

  Assume that you receive a file that contains historical data. The name of the file identifies the date of the historical data. For example, a filename of **19960424** indicates that the data was produced on April 24, 1996. To map this date to **RptDate**, the TODATETIME function could be used with the RIGHT and GETRESOURCENAME functions.

Related functions

- CURRENTDATE
- CURRENTDATETIME
- CURRENTTIME
- TEXTTODATE
- TEXTTOTIME

Chapter 9. Error handling functions

CONTAINSERRORS

The CONTAINSERRORS function tests a valid object to see whether it contains any objects in error.

Syntax:
CONTAINSERRORS (single-object-expression)

Meaning:
CONTAINSERRORS (object_to_test)

Returns:
True or false

The CONTAINSERRORS function returns ″true″ if any object contained in *object_to_test* is in error; it returns ″false″ if the *object_to_test* is completely valid.

The input object, itself, is a *valid* input object. This function does *not* evaluate for invalid objects. Therefore, if you have map rule:

CONTAINSERRORS (Invoice:InputFile)

and **Invoice**[1] is valid, **Invoice**[2] is invalid, and **Invoice**[3] is valid, then CONTAINSERRORS evaluates only twice-once for each *valid* instance of **Invoice**.

Examples

- Msg (s) = IF (CONTAINSERRORS (Msg:MailBag) &
  Type:Msg:MailBag = ″PRIORITY″ ,
  Msg:MailBag,
  ″none″)

  In this example, if **Msg:MailBag** contains any object in error and **Type:Msg:MailBag** has a value of ″PRIORITY″, **Msg:MailBag** is mapped; otherwise, ″none″ is returned for this occurrence of **Msg**. This map rule returns all valid messages (**Msg**) that contain errors with a **Type** of ″PRIORITY″.

Related functions

- ISERROR
- REFORMAT

FAIL

You can use the FAIL function to abort a map based on map or application specific logic. ″none″, aborts the map, and returns a text string to be reported as the map completion error message.

Syntax:
FAIL (single-text-expression)

Meaning:
FAIL (message_to_return)

Returns:
″None″

The FAIL function returns ″none″ to the output to which the function is assigned, aborts the map, and returns *message_to_return* as the map completion error message included in the execution audit. The map return code will be ″30″, indicating that the map failed through the FAIL function.

Examples

- AcctID = EITHER (LOOKUP (CustomerID:.:Xref, MyKey:.:Xref = ID:.:Input) , FAIL (″Unknown Customer (″ + ID:.:Input + ″). Processing Terminated.″))

  In this example, the FAIL function is being used in conjunction with the EITHER function to conditionally fail the map if a record in the customer cross-reference file does not exist for a given **CustomerID**.

  For example, the **ID** in **Input** is ABC123. If the LOOKUP succeeds, the **CustomerID** result is assigned to **AcctID** and the map continues.

  If the LOOKUP fails, **AcctID** is assigned a value of ″none″, the map aborts, and the message Unknown Customer (ABC123). Processing Terminated.is written to the execution audit log.
- Message = VALID (RUN (″Map1Msg.mmc″, ″-AE -OMMSMQ1B `-QN .\aqueue -CID 2001′″),
  FAIL (″Failure on RUN (″ + TEXT (LASTERRORCODE ( )) + ″):″ + LASTERRORMSG ( )))

  In this example, the FAIL function is being used in conjunction with the VALID, LASTERRORCODE, and LASTERRORMSG functions to fail (abort) the map if the map executed by the RUN function (**Map1Msg.mmc**) fails. In this example, the map fails and returns the error code and error message reported by the RUN function using the LASTERRORCODE and LASTERRORMSG functions.

  If **Map1Msg** fails because one or more of its inputs was invalid, **Message** is assigned a value of ″none″. The map aborts and the following message is reported in the execution audit log:

  ″Failure on RUN (8): One or more inputs was invalid.″

Related functions

- LASTERRORCODE
- LASTERRORMSG
- VALID

ISERROR

The ISERROR function tests an object to see if it is in error

You can use ISERROR to output your data in exactly the same order as it occurs in your input, both the valid data and the data in error. You can also use ISERROR to produce error messages for bad data in the same file in which you map your good data.

Syntax:
ISERROR (single-object-name)

Meaning:
ISERROR (object_to_test)

Returns:
"True" or "false"

ISERROR returns "true" when *object_to_test* is in error and returns "false" when *object_to_test* is completely valid.

Examples

- InfoRec (s) = IF (ISERROR (Record:SomeFile), "Bad --> " + REJECT (Record:SomeFile), "Ok --> " + TEXT (Record:SomeFile))

  In this example, ISERROR is used to produce a report for *all* **Record** objects in **SomeFile**. If the record is in error, the **InfoRec** will have the text Bad --> followed by the data from the input **Record**. If the record is valid, the **InfoRec** will have the text Ok --> followed by the data from the input **Record**, such as:

```
Ok  --> SZ-68839,486 Upgrade Microprocessor,186.86,100,W200
Bad --> MK-19309,,369.43,417,W100
Ok  --> KL-20349,PCMCIA Network Adaptor,174.82,29,N300
Ok  --> WP-37679,AC Adaptor,39.48,245,E100
Bad --> IL-39890,8MB Memory PCMCIA,390.48,0,S100
```

Related functions

- CONTAINSERRORS

ONERROR

Use ONERROR in component rules to add user-defined error messages to the data section of the audit log.

The ONERROR function adds a user-defined error message to the data section of the audit log. This function is relevant only in Type Designer Component rules.

.

Syntax:
ONERROR (single-condition-expression, single-text-expression)

Meaning:
ONERROR (condition_to_evaluate , message_to_display)

Returns:
"True" or "false"

If *condition_to_evaluate* evaluates to "true", the function returns "true".

If *condition_to_evaluate* evaluates to "false", the function returns "false".

If data audit is enabled and the object to which the component rule applies will result in a failed component rule message (status E09), *message_to_display* is written to the data audit section of the audit log with an entry type of U, representing user-defined.

If the failed component rule message (E09) appears in the data section of the audit log, one or more user-defined error messages can also be included in the data section of the audit log. If a component rule has one ONERROR function, one user-defined error message or none is included in the audit log. If a component rule has two ONERROR functions, two user-defined error messages at most are included in the audit log, and so on.

Examples

- Here is a component rule without ONERROR:

  Claim Date Field > Accident Date Field &

  Claim Date Field < ADDDAYS (Accident Date Field, 365)

  If the component rule fails, the data section of the audit log shows the following information:

```
<DataLog>
<input card="1">
 <object      ...      status="E07">InsuranceClaim</object>
 <object      ...      status="E09">Claim Date Field</object>
                                <Text>980725</Text>
 <object      ...      status="E07">InsuranceClaim</object>
 <object      ...      status="E09">Claim Date Field</object>
                                <Text>990526</Text>
</input>
</DataLog>
```

Status code E09 for the **Claim Date Field** indicates that the data for that object failed its component rule, but does not provide any further detail.

- Using ONERROR, one or more error messages can be added to the data section of the audit log to provide more information as to how or why the component rule failed. For example,

  ONERROR (Claim Date Field > Accident Date Field ,
  ″Claim date before accident.″) &

  ONERROR (Claim Date Field < ADDDAYS (Accident Date Field , 365) ,
  ″Claim is more than one year old.″)

  If the component rule fails and ONERROR is used, the data section of the audit log can show the following messages:

```
<DataLog>
<input card="1">
 <object  ...  status="E07">InsuranceClaim</object>
 <object  ...  status="E09">Claim Date Field</object>
                           <Text>980725</Text>
                           <User>Claim date before accident.</User>
 <object  ...  status="E07">InsuranceClaim</object>
 <object  ...  status="E09">Claim Date Field</object>
                           <Text>990526</Text>
                           <User>Claim is more than one year old.</User>
</input>
</DataLog>
```

REJECT

The REJECT function returns the content of an object in error as a text item. Use REJECT in conjunction with the restart attribute. You can use the REJECT function in map rules only, not in component rules.

See the Type Designer and Map Designer documentation for information about the restart attribute.

Syntax:

REJECT (series-object-expression)

Meaning:

REJECT (series_to_look_for_bad_objects)

Returns:

A series text item

REJECT evaluates to a series of text items consisting of all the input series members in error.

Examples

- REJECT (Record:File)

  This example extracts all **Records** that are in error.
- IF (COUNT (REJECT (Msg IN Batch))) = 0, ″OK″, ″ERROR″)

  In this example, the total number of invalid (rejected) Msg objects is counted. If the total number of invalid Msg objects is equal to zero, it indicates that there were no invalid Msg objects counted and a message of OK results. Otherwise, a message of ERROR results.

Related functions

- CONTAINSERRORS
- ISERROR
- VALID

VALID

You can use the VALID function to perform conditional processing based on whether an external interface function executes successfully.

VALID returns the result of the first argument if it is valid; otherwise, returns the second argument.

Syntax:

VALID (single-text-expressions , single-general-expression)

Meaning:

VALID (function_that_can_fail , return_value_if_function_fails)

Returns:

A single text expression

VALID returns the result of the evaluation of *function_that_can_fail* if it is valid. If the function fails, VALID returns *return_value_if_function_fails*.

The following functions can fail:

• DBLOOKUP	• GET
• DBQUERY	• PUT
• DDEQUERY	• RUN
• EXIT	

Examples

- SomeObject = VALID (RUN (″mymap.mmc″ , ″-OF1 mydata.txt″) , FAIL (″My RUN failed!″))

  If the RUN function returns an error return code, the VALID functions returns ″none″, the map aborts, and the message ″My RUN failed!″ is reported under ″Execution Summary″ in the execution audit log.

Related functions

- DBLOOKUP
- DBQUERY
- DDEQUERY

Chapter 10. External interface functions

DBLOOKUP

The DBLOOKUP function executes an SQL statement against a database. The SQL statement can be any permitted by your database management system or ODBC driver.

When the DBLOOKUP function is used in a map, the default **OnSuccess** action is adapter-specific. The default **OnFailure** action is to rollback any changes made during map processing. The default **Scope** will be integral unless the map is defined to run in bursts (which is the case when one or more inputs have the **FetchAs** property set to **Burst**).

There are two ways to specify arguments for DBLOOKUP.

You can use DBLOOKUP to execute an SQL statement when you want to execute a SELECT statement to retrieve a specific column value in a large table in a database using the value of another input, rather than defining the entire table as an input card and using the LOOKUP, SEARCHDOWN, or SEARCHUP functions.

You can use DBLOOKUP to execute an SQL statement when you want to execute a SELECT statement to retrieve a specific column value from a table or database that might vary based on a parameter file. Using Meaning 2 of the DBLOOKUP function allows these parameters to be dynamically specified at run time.

Syntax:

DBLOOKUP (single-text-expression , single-text-expression , [single-text-literal])

Meaning:

1. DBLOOKUP (SQL_statement , mdq_filename, database_name)
2. DBLOOKUP (SQL_statement , parameters)

Returns:

A single text item

The DBLOOKUP function returns the results of the query in the same format as a query specified for a map input card, except that it does not include the last carriage return/linefeed. Because this information is removed, it is easier to make use of a single value extracted from a database.

Arguments for meaning 1

DBLOOKUP (SQL_statement , mdq_filename , database_name)

- **SQL_statement**

  The first argument is an SQL statement as a text string. This can be any valid SQL statement permitted by your database management system and supported by your database-specific driver. In addition to a fixed SQL statement, this argument can be a concatenation of text literals and data objects, enabling the concatenation of data values into your SQL statement.

- **mdq_filename**

The second argument is the name of a database query file (**.mdq**) produced by the Database Interface Designer. It contains the definition of the database that the SQL statement is to be executed against. If the **.mdq** file is in a directory other than the directory of the map, the path must be specified.

Note: The **.mdq** file is accessed at map build time and is not needed at runtime.

- **database_name**

  The third argument is the name of a database in the database query file (**.mdq**) as defined in the Database Interface Designer.

  If used in this way, both the **.mdq** filename and database name must be literals.

Arguments for meaning 2

DBLOOKUP (SQL_statement , parameters)

- **SQL_statement**

  The first argument is an SQL statement as a text string. This can be any valid SQL statement permitted by your database management system and supported by your database-specific driver. In addition to a fixed SQL statement, this argument can be a concatenation of text literals and data objects, enabling the concatenation of data values into your SQL statement.

- **parameters**

  The second argument is a set of parameters, either:

  – -MDQ *mdqfilename* -DBNAME *dbname*

  -or-

  – -DBTYPE *database_type* [*database specific parameters*]

  The keyword -MDQ is followed by the name of the database query file (**.mdq**) produced by the Database Interface Designer. This **.mdq** file contains the definition of the database. If the **.mdq** file is in a directory other than the directory of the map, the path must be specified. The **.mdq** filename is followed by the keyword -DBNAME and the database name as specified in the Database Interface Designer.

  Using this syntax, the **.mdq** file is accessed at runtime and must be present.

  The keyword -DBTYPE is followed by a keyword specifying the database type (for example, ODBC or ORACLE) followed, optionally, by database-specific parameters.

  This syntax does not use an **.mdq** file, because the database-specific parameters provide the information required to connect to the database. Refer to the Resource Adapters documentation for detailed information on the database-specific parameters that can be specified.

When used with Meaning 2, DBLOOKUP must conform to these rules:

- All keywords (for example, -DBTYPE) can be upper or lowercase, but not mixed.
- A space is required between the keyword and its value (for example, -DBTYPE ODBC).
- The order of the keywords is not important.

  All database-specific parameters are optional.

Examples

Assume that you have a table named ″PARTS″ that contains the following data:

PART_NUMBER	PART_NAME
1	1/4″ x 3″ Bolt
2	1/4″ x 4″ Bolt

Assume that this database has been defined using the Database Interface Designer in a file named **mytest.mdq** and that the name of the database, as specified in the **.mdq** file, is **PartsDB**.

- DBLOOKUP (″SELECT PART_NAME from PARTS where PART_NUMBER =1″, ″mytest.mdq″, ″PartsDB″)

  Returns: ¼″ x 3″ Bolt

Using Meaning 2, you can specify the DBLOOKUP this way:

- DBLOOKUP(″SELECT PART_NAME from PARTS where PART_NUMBER =1″, ″-MDQ mytest.mdq -DBNAME PartsDB″)

  where both the **.mdq** file name and database name is specified.

Using Meaning 2, you can also specify the database type and the appropriate database-specific parameters:

- DBLOOKUP(″SELECT PART_NAME from PARTS where PART_NUMBER =1″ , ″-DBTYPE ORACLE -CONNECT MyDB -USER janes ″)

Related functions

- DBQUERY
- EXTRACT
- FAIL
- LASTERRORCODE
- LASTERRORMSG
- LOOKUP
- SEARCHDOWN
- SEARCHUP
- VALID

For more examples using the DBLOOKUP function, see the Database Interface Designer documentation.

DBQUERY

The DBQUERY function executes an SQL statement against a database. The SQL statement can be any permitted by your database management system or ODBC driver.

When the DBQUERY function is used in a map, the default **OnSuccess** action is adapter-specific. The default **OnFailure** action is to rollback any changes made

during map processing. The default **Scope** will be integral unless the map is defined to run in bursts (which is the case when one or more inputs have the **FetchAs** property set to **Burst**).

There are two ways to specify the arguments for DBQUERY. You can use DBQUERY [*Meaning 1*] to execute an SQL statement when you want to look up information in a database using a parameterized query that is based on another value in your data. If your SQL statement is a SELECT statement, the DBQUERY function may be used in conjunction with the RUN function to issue dynamic SELECT statements whose results can be used as input to another map.

You can also use the DBQUERY function [*Meaning 2*] to execute an SQL statement when the database, table, or other database parameters might vary; perhaps being supplied by a parameter file.

Syntax:

```
DBQUERY (single-text-expression , single-text-expression ,
 [ single-text-literal ] )
```

Meaning:

1. DBQUERY (SQL_statement , mdq_filename, database_name)
2. DBQUERY (SQL_statement , parameters)

Returns:

A single text item

If your SQL statement is a SELECT statement, the results of the query in the same format as a query specified as a map input card, including row delimiters and terminators, and so on.

If your SQL statement is anything other than a SELECT statement, ″none″.

Arguments for meaning 1

DBQUERY (SQL_statement, mdq_filename , database_name)

- **SQL_statement**

  The first argument is an SQL statement as a text string. This can be any valid SQL statement that is permitted by your database management system and supported by your database-specific driver. In addition to a fixed SQL statement, this argument can be a concatenation of text literals and data objects, enabling the concatenation of data values into your SQL statement.

- **mdq_filename**

  The second argument is the name of a database query file (**.mdq**) produced by the Database Interface Designer. It contains the definition of the database that the SQL statement is to be executed against. If the **.mdq** file is in a directory other than the directory of the map, the path must be specified.

  Note: The **.mdq** file is accessed at map build time and is not needed at runtime.

- **database_name**

  The third argument is the name of a database in the database query file (**.mdq**) as defined in the Database Interface Designer.

  If used in this way, both the **.mdq** filename and database name must be literals.

Arguments for meaning 2

DBQUERY (SQL_statement , parameters)

- The first argument is an SQL statement as a text string. This can be any valid SQL statement that is permitted by your database management system and supported by your database-specific driver. In addition to a fixed SQL statement, this argument can be a concatenation of text literals and data objects, enabling the concatenation of data values into your SQL statement.
- The second argument is a set of parameters, either:
  - -MDQ mdqfilename -DBNAME dbname

    -or-
  - -DBTYPE database_type [database specific parameters]

    The keyword -MDQ is followed by the name of the database query file (**.mdq**) produced by the Database Interface Designer. This **.mdq** file contains the definition of the database. If the **.mdq** file is in a directory other than the directory of the map, the path must be specified. The **.mdq** filename is followed by the keyword -DBNAME and the database name as specified in the Database Interface Designer.

    Note: Using this syntax, the **.mdq** file is accessed at runtime and must be present.

    The keyword -DBTYPE is followed by a keyword specifying the database type (for example, ODBC or ORACLE) followed, optionally, by database-specific parameters.

    Note: This syntax does not use an **.mdq** file, because the database-specific parameters provide the information required to connect to the database. Refer to the appropriate database adapter documentation for detailed information about database-specific parameters.

When used with Meaning 2, DBQUERY must conform to these rules:

- All keywords (for example, -DBTYPE) can be upper or lower case, but not mixed.
- A space is required between the keyword and its value (for example, -DBTYPE ODBC).
- The order of the keywords is not important.

  All database-specific parameters are optional.

Examples

Assume that you have a table named ″PARTS″ that contains the following data:

PART_NUMBER	PART_NAME
1	1/4″ x 3″ Bolt
2	1/4″ x 4″ Bolt

Also assume that this database has been defined using the Database Interface Designer in a file named **mytest.mdq** and that the name of the database, as specified in the **.mdq** file, is **PartsDB**.

```
DBQUERY ( "SELECT * from PARTS" ,  "mytest.mdq" ,  "PartsDB" )
```

Returns 1|¼″ x 3″ Bolt<cr><lf>2|¼″ x 4″ Bolt<cr><lf>

where <cr><lf> is a carriage return followed by a line feed.

Using Meaning 2, you can also specify the DBQUERY this way:

```
DBQUERY ( "SELECT * from PARTS" ,  "-MDQ mytest.mdq -DBNAME PartsDB" )
```

where both the **.mdq** file name and database name are specified.

Or, specify it this way, using Meaning 2 by specifying the database type and the appropriate database-specific parameters:

```
DBQUERY ( "SELECT * from PARTS" ,  "-DBTYPE ORACLE -CONNECT MyDB -USER
janes" )
```

Assume that you have an input file containing one order record. To map that order to another proprietary format, you also have a parts table with pricing information for every part for every customer, a very large table. Rather than using the entire parts table as the input to your map, you might use the RUN function with a DBQUERY to dynamically select only those rows from the parts table corresponding to the customer in the order file, as follows:

```
RUN ( "MapOrder.MMC" ,
  "IE2" + DBQUERY ( "SELECT * FROM Parts WHERE CustID = "
 + CustomerNo:OrderRecord:OrderFile + " ORDER BY PartNo" ,
"PartsDB.MDQ", "PartsDatabase" ) )
```

Related functions

- DBLOOKUP
- EXTRACT
- FAIL
- LASTERRORCODE
- LASTERRORMSG
- LOOKUP
- SEARCHUP
- SEARCHDOWN
- VALID

DDEQUERY

The DDEQUERY function allows you to interface to other Windows applications such as Trading Partner PC, Excel, and so forth, provided that certain criteria are met. For example, if you receive an Excel spreadsheet file, you must have the appropriate version of the Excel application installed (that is compatible with the file received) and the application must be open.

Syntax:

DDEQUERY (single-text-expression , single-text-expression, single-text-expression)

Meaning:

DDEQUERY (application_name , topic , text)

Returns:

A single text item from an application

Examples

- DDEQUERY (″excel″ , ″[MKTPRICE.XLS]Sheet1″, ″R8C1:R14C3″)

  In this example, DDEQUERY is used to get data from an Excel spreadsheet. The third argument, **R8C1:R14C3**, specifies the location of the data in the spreadsheet. (In Excel, the 8th row, 1st column to the 14th row, 3rd column is A8:C14.) The content of this spreadsheet range is returned as a single text item.

  This example assumes that the application, the map, and the spreadsheet all reside in the same directory. If they are not in the same directory you must add the path. For example:

  DDEQUERY (″excel″ , ″c:\spreadsheet[MKTPRICE.XLS]Sheet1″ , ″R8C1:R14C3″)
- DDEQYUERY (″tppc″,″PartnerX″,″BGyourEDIode″)

  In this example, DDEQUERY is used as a request to Trading Partner PC.

Related functions

• EXIT	• LASTERRORMSG
• FAIL	• PUT
• GET	• RUN
• LASTERRORCODE	• VALID

EXIT

The EXIT function allows you to interface with a function in an external library or application.

You can use EXIT when you need information from an existing function in a library or a program or when you need to use a general function that is not available.

Depending on the execution platform, there are two different methods for the EXIT function: 1) the library method and 2) the program method. The program method is not supported on Windows platforms.

Syntax:

EXIT (single-text-expression, single-text-expression, single-text-expression)

Meaning:

1. EXIT (library_name, function_name, input_to_the_function)
2. EXIT (program_name , command_line_arg1 , command_line_arg2)

Returns:

A single text item

Meaning 1 - library method

At runtime, the *function_name* function will execute in the library specified by *library_name* passing *input_to_the_function* as a text string. The result of *function_name* is returned as a text item by means of **lpep-** → **lpdataFromApp**.

Set **lpep-** → **nReturn** equal to 0 if the function is to succeed or set it equal to 1 to fail.

For detailed information on the requirements of the library function that is executed by the EXIT, see ″Implementing a library EXIT function″.

AIX Platform

On the AIX platform, the shared library that contains *library_function* must also contain an *entry_point* function. The *entry_point* function must be prototyped in the same manner as *library_function*. The server will invoke the *entry_point* function before invoking *library_function*. The server passes the name of the *library_function* to the *entry_point* function by way of the *szFile* EXITPARAM structure member. The *entry_point* function must then determine the address of the *library_function* and pass this address back to the server by way of the *lpv* EXITPARAM structure member. The server will then use the contents of *lpv* as an address to invoke the *library_function*.

The following example depicts a shared library called **mcshex.so** which contains the *entry_point* function and the *library_function*.

```
   #include <stdio.h>
#include <string.h>
#include "runmerc.h"
void AIXEntry(LPEXITPARAM);
void bin2hex(LPEXITPARAM);
unsigned int i;
char *syms[] = { "bin2hex", "AIXEntry" };
void *adr[]  = { (void *)bin2hex, (void *)AIXEntry};
void AIXEntry(LPEXITPARAM lpInputStruct)
{
    printf("AIXEntry called\n");
    for (i = 0; i < (sizeof(syms)) / (sizeof(char *)); i++)
       if ( !(strcmp(lpInputStruct->szFile, syms[i])) ) {
           lpInputStruct->lpv = adr[i];
           break;
       }
  }
void bin2hex(LPEXITPARAM lpInputStruct)
  {
    printf("bin2hex called\n");
printf("argument to library function: %s\n",lpInputStruct->lpszCmdLine);
lpInputStruct->nReturn = 0;
}
```

To build this shared library, run the following commands:

cc -c share1.c

cc -o mcshex.so share1.o -bE:shrsub.exp -bM:SRE -eAIXEntry

Meaning 2 - program method

The program method of the EXIT function is *not* supported on Windows platforms.

At execution time, the program specified by *program_name* executes and passes the concatenation of *command_line_arg1* + ″ ″ + *command_line_arg2* as a text string.

Whatever is returned by *program_name* to the standard output device is returned as text.

Examples

- EXIT (*program_name* , *command_line_arg1* , *command_line_arg2*)

  Returns a text string from the function or application that is executed. If the EXIT function is not available for a particular platform, EXIT returns ″none″.
- EXIT (″mydll.dll″ , ″myfunction″ , ″12″)

  This Windows library example passes the value 12 to **myfunction**, a function in **mydll.dll.** The value of the item returned depends on what **myfunction** does with the 12 passed to it.
- IF (EXIT (″mylib.sl″ , ″ckCust″ , CustID Column:Row:DB) = ″OK″ , MapKnownCust (Row:DB) , MapUnknownCust (Row:DB))

  Similarly, this UNIX library example passes the value of **CustID** Field to ckCust, a function in a UNIX shared library called **mylib.sl**. If the value returned by ckCust is OK, a functional map is called to map **Row** for a known customer. Otherwise, another functional map is executed to map **Row** for an unknown customer.
- EXIT (″pwd″ , ″ ″ , ″ ″)

  This UNIX program example executes the UNIX print working directory (pwd) command to determine the current directory. The name of the current working directory is then returned as a text string. Notice that although the pwd command does not require additional command line arguments, *command_line_arg1* and *command_line_arg2* must be included as a space enclosed in quotes (″ ″).
- TEXTTONUMBER ((EXIT (″GetIncome″ , Applicant:Form , ″*Mortgage″))

  This program example passes the value of Applicant concatenated to the text literal *Mortgage to an application called GetIncome. The result is converted to a number.

Related functions

- DDEQUERY
- FAIL
- GET
- LASTERRORCODE
- LASTERRORMSG
- PUT
- RUN
- VALID

GET

You can use the GET function to retrieve data using one of the source adapters, such as a messaging system, a database, a file, and so forth, within the course of your map.

When the GET function is used in a map, the default **OnSuccess** action is adapter-specific. The default **OnFailure** action is to rollback any changes made during map processing. The default **Scope** will be integral unless the map is defined to run in bursts (which is the case when one or more inputs have the **FetchAs** property set to **Burst**).

GET can also be used for adapters that support request/reply, such as Roma.

Syntax:

```
GET (single-text-expression , single-text-expression
 [ , single-text-expression ])
```

Meaning:

```
GET (adapter_alias, adapter_commands
 [ , data_request_to_send_to_adapter ])
```

Returns:
A single character text item

GET returns the data that is returned by the source adapter. The adapter identified by *adapter_alias* is called using the specified *adapter_commands* and passing *data_request_to_send_to_adapter* as data to the adapter. See the Resource Adapters documentation.

To identify whether the function was successful, you can generally use the VALID function with the GET function. However, for certain adapters such as E-mail and FTP, the success state is not known until after map completion and using the VALID function in such cases might consistently return ″success″.

Examples

- Reply (s) = GET (″ROMA″, ″-BE Orders22A -MID 32001 -REQ 2″, PACKAGE (Request Object:.:Input)

  In this example, the GET function could be used in a request/reply mode to get information from a Roma messaging system.
- Acct# = GET (″DB″ , ″-dbtype ORACLE -connect shasta -user rjc -pw vm70″ + ″SELECT Acct# FROM CustMaster WHERE CustID = ′″ + SenderID Field:Identification Segment:Msg + ″′″)

  In this example, the GET function could be used instead of a DBLOOKUP or DBQUERY function to retrieve data from a database from within in a map rule.

Related functions

- FAIL
- LASTERRORCODE
- LASTERRORMSG
- PUT
- VALID

PUT

The PUT function passes data to a target adapter.

Use PUT to route data within your map using one of the target adapters such as to a messaging system, a database, a file, and so forth.

When the PUT function is used in a map, the default **OnSuccess** action is adapter-specific. The default **OnFailure** action is to rollback any changes made during map processing. The default **Scope** will be integral unless the map is defined to run in bursts (which is the case when one or more inputs have the **FetchAs** property set to **Burst**).

Syntax:

PUT (single-text-expression , single-text-expression, single-text-expression)

Meaning:

PUT (adapter_alias , adapter_commands, data_to_send_to_adapter)

Returns:

″None″

The adapter identified by *adapter_alias* is called using the specified *adapter_commands* and passing *data_to_send_to_adapter* as data to the adapter.

To identify whether the function was successful, you can generally use the VALID function with the PUT function. However, for certain adapters such as E-mail and FTP, the success state is not known until after map completion and using the VALID function in such cases might consistently return ″success″.

Examples

This example illustrates a mapping situation in which a set of messages is being produced in the output (output card #1). However, the ultimate target for these messages is a message queue and the messages need to be placed on the queue one at a time.

To accomplish this, the first output card builds the set of messages and its **Target** is defined as **Sink**. So, this set of messages is constructed in memory, but is discarded after the map completes. A second output card, uses the following **PUT** function to put each output message (from output card# 1), individually, on the output queue.

Message(s) = PUT (″MQS″, ″-QMN myqueuemgr -QN chips_queue -T″, PACKAGE (PaymentMessage:CHIPS_Payment_Message))

- The first argument, MQS, identifies that the data is to be routed using the IBM WebSphere MQ (server) adapter.
- The second argument, ″-QMN myqueuemgr -QN chips_queue -T″, provides the adapter commands needed by the adapter to put the message on the queue.
- The third argument, PACKAGE, passes the data that is to be sent as the body of the message.

Related functions

- FAIL
- GET
- LASTERRORCODE
- LASTERRORMSG
- VALID

RUN

The RUN function allows you to execute another compiled map from a component or map rule.

You can use RUN to dynamically name source and/or destination files or to dynamically pass data to a map. You can also use the RUN function to split the output data into separate files based on some value in the input.

Syntax:

RUN (single-text-expression [, single-text-expression])

Meaning:

RUN (map_to_run [, command_option_list])

Returns:

A single text item

The first argument, *map_to_run*, is an expression identifying the name of the compiled map (**.mmc**) to be run.

The *command_option_list* argument is an optional argument that you can use to specify execution commands applicable to the map to be run. *Command_option_list* is a text item containing a series of execution commands separated by a space. Any execution command can be used as part of the *command_option_list* argument. For example, you can send data to another map by using the echo command option (`-IEx`).

See the Execution Commands documentation for a list of command options.

The result of the RUN function depends on the command options in *command_option_list*.

Echo command option

- If you use the Echo command option for an output card, the data from that card will be passed back as a text item to the object in the map from which it was run.
- If you use the Echo command option for more than one output card, the data from all echoed cards will be concatenated together and passed back as a text-item to the object in the map from which it was run.
- If you do not use the Echo command option, the return code indicating the status of the map that was run will be passed back to the object in the map from which it was run.

Examples

- RUN ("MyMap" , "-IE1s502 " + Invoice:File + " -OF1 *install_dir*\" + CustomerID:Invoice)

  This example runs the **MyMap** map, sending 502-byte fixed-size **Invoice** data as the data source for input card **1**, overriding the filename of output card **1** based on **Customer** data and returns the map return code as the result.
- RUN ("GetDbOpt" , " ")

  This example runs the **GetDbOpt** map (with no command options specified) and returns the map return code as the result.
- RUN ("DoOneSet" , "-A -GR" +
  ECHOIN(1 , Set:InFile)) +
  " -OF1 OUT_SET." + NUMBERTOTEXT (INDEX ($)) +
  " -OE2")

  This example runs the **DoOneSet** map. Command options include the following:
  - "-A -GR"

    The **DoOneSet** map will produce no **Audit Log** and restrictions will be ignored.
  - ECHOIN(1 , Set:InFile)

The ECHOIN function creates the -IE command option for echoing the data represented by Set:InFile to input **1** of the RUN map.

– ″ -OF1 OUT_SET.″ + NUMBERTOTEXT (INDEX ($))

  The output file for card **1** of the **DoOneSet** map will be called ″OUT_SET.″ plus a sequence number based on the index of the Set in **InFile**. For example, the first output set will be OUT_SET.1, and so forth.

– ″ -OE2″

  Using the output echo command option, the data built for output card **2** of the **DoOneSet** map will be returned as the result of the RUN function.

  An alternative to using the ECHOIN function shown above is using the long version. For example, replace ECHOIN(1 , Set:InFile) with:

  ″ -IE1S″ + NUMBERTOTEXT (SIZE (Set:InFile) + ″ ″ + TEXT (Set:InFile)

  Using the Echo input command option (**-IE**) with the sizing method (**Sn**) or the ECHOIN function, one **Set** object of **InFile** is passed to input card **1** for the **DoOneSet** map.

Related functions

- DDEQUERY

Chapter 11. Inspection functions

ABSENT

The ABSENT function tests for the absence of an object.

Syntax:
ABSENT (single-object-expression)

Meaning:
ABSENT (object_to_test)

Returns:
True or false

ABSENT returns ″true″ if the object_to_test evaluates to ″none″. If the object_to_test does not evaluate to ″none″, the function returns ″false″.

Examples

- You can use this function to map an object only if another object is absent. For example, you might want to map **BillTo** information to the **ShipTo** fields if the **ShipToName** is absent.
- ABSENT (AreaCode:Phone)

  This example evaluates to ″true″ when **AreaCode:Phone** evaluates to ″none″ or evaluates to ″false″ when **AreaCode:Phone** does not evaluate to ″none″.

Related Functions

- PRESENT

CONTAINSERRORS

The CONTAINSERRORS function tests a valid object to see whether it contains any objects in error.

Syntax:
CONTAINSERRORS (single-object-expression)

Meaning:
CONTAINSERRORS (object_to_test)

Returns:
True or false

The CONTAINSERRORS function returns ″true″ if any object contained in *object_to_test* is in error; it returns ″false″ if the *object_to_test* is completely valid.

The input object, itself, is a *valid* input object. This function does *not* evaluate for invalid objects. Therefore, if you have map rule:

CONTAINSERRORS (Invoice:InputFile)

and **Invoice**[1] is valid, **Invoice**[2] is invalid, and **Invoice**[3] is valid, then CONTAINSERRORS evaluates only twice-once for each *valid* instance of **Invoice**.

Examples

- Msg (s) = IF (CONTAINSERRORS (Msg:MailBag) &
  Type:Msg:MailBag = ″PRIORITY″ ,
  Msg:MailBag,
  ″none″)

  In this example, if **Msg:MailBag** contains any object in error and **Type:Msg:MailBag** has a value of ″PRIORITY″, **Msg:MailBag** is mapped; otherwise, ″none″ is returned for this occurrence of **Msg**. This map rule returns all valid messages (**Msg**) that contain errors with a **Type** of ″PRIORITY″.

Related functions

- ISERROR
- REFORMAT

ISALPHA

You can use ISALPHA when you need to know whether a text string is all alphabetic characters.

Syntax:

ISALPHA (single-text-expression)

Meaning:

ISALPHA (text_to_test)

Returns:

This function evaluates to a Boolean ″true″ or ″false″ and should only be used as a conditional expression within a logical function.

The ISALPHA function tests a text object to see if it is contains all alphabetic characters.

If *text_to_test* contains only alphabetic characters (for example, A-Z and a-z), ISALPHA evaluates to ″true″.

If *text_to_test* contains other than just alphabetic characters, ISALPHA returns ″false″.

Examples

- IF(ISALPHA (″AnywhereUSA″))
  Returns ″true″
- IF(ISALPHA (″Anywhere USA″))
  Returns ″false″
- IF(ISALPHA (″Mr. Brown″))
  Returns ″false″

Related functions

• ISLOWER	• LEAVEALPHANUM
• ISNUMBER	• LEAVENUM
• ISUPPER	• LEAVEPRINT
• LEAVEALPHA	

ISERROR

The ISERROR function tests an object to see if it is in error

You can use ISERROR to output your data in exactly the same order as it occurs in your input, both the valid data and the data in error. You can also use ISERROR to produce error messages for bad data in the same file in which you map your good data.

Syntax:
ISERROR (single-object-name)

Meaning:
ISERROR (object_to_test)

Returns:
"True" or "false"

ISERROR returns "true" when *object_to_test* is in error and returns "false" when *object_to_test* is completely valid.

Examples

- InfoRec (s) = IF (ISERROR (Record:SomeFile), "Bad --> " + REJECT (Record:SomeFile), "Ok --> " + TEXT (Record:SomeFile))

  In this example, ISERROR is used to produce a report for *all* **Record** objects in **SomeFile**. If the record is in error, the **InfoRec** will have the text Bad --> followed by the data from the input **Record**. If the record is valid, the **InfoRec** will have the text Ok --> followed by the data from the input **Record**, such as:

```
Ok  --> SZ-68839,486 Upgrade Microprocessor,186.86,100,W200
Bad --> MK-19309,,369.43,417,W100
Ok  --> KL-20349,PCMCIA Network Adaptor,174.82,29,N300
Ok  --> WP-37679,AC Adaptor,39.48,245,E100
Bad --> IL-39890,8MB Memory PCMCIA,390.48,0,S100
```

Related functions

- CONTAINSERRORS

ISLOWER

The ISLOWER function tests a text object to see if it contains all lowercase alphabetic characters.

Syntax:
ISLOWER (series-text-expression)

Meaning:
ISLOWER (text_to_test)

Returns:
This function evaluates to a Boolean "true" or "false" and should only be used as a conditional expression within a logical function.

If *text_to_test* contains only lowercase alphabetic characters (for example, a-z), ISLOWER evaluates to "true".

If *text_to_test* contains other than lowercase alphabetic characters, ISLOWER returns "false".

Examples

- IF(ISLOWER (″company″))
  Returns ″true″
- IF(ISLOWER (″pots and pans″))
  Returns ″false″
- IF(ISLOWER (″Andrew″))
  Returns ″false″

Related functions

• ISALPHA	• LEAVEALPHANUM
• ISNUMBER	• LEAVENUM
• ISUPPER	• LEAVEPRINT
• LEAVEALPHA	

ISNUMBER

The ISNUMBER function tests a text object to determine whether it contains all numeric characters.

Syntax:
ISNUMBER (single-text-expression)

Meaning:
ISNUMBER (text_to_test)

Returns:
This function evaluates to a Boolean ″true″ or ″false″ and should only be used as a conditional expression within a logical function.

If *text_to_test* contains only digits (for example, 0-9), ISNUMBER evaluates to ″true″.

If *text_to_test* contains other than digits, ISNUMBER evaluates to ″false″.

Examples

- IF(ISNUMBER(″1″)
  Returns ″true″

Related functions

• ISALPHA	• LEAVEALPHANUM
• ISLOWER	• LEAVENUM
• ISUPPER	• LEAVEPRINT
• LEAVEALPHA	

ISUPPER

The ISUPPER function tests a text object to determine whether it contains all uppercase alphabetic characters.

Syntax:

ISUPPER (series-text-expression)

Meaning:

ISUPPER (text_to_test)

Returns:

This function evaluates to a Boolean ″true″ or ″false″ and should only be used as a conditional expression within a logical function.

If *text_to_test* contains only uppercase alphabetic characters (for example, A-Z), ISUPPER evaluates to ″true″.

If *text_to_test* contains other than uppercase alphabetic characters, ISUPPER returns ″false″.

Examples

- IF(ISUPPER (″BOMBAY″))

  Returns ″true″
- IF(ISUPPER (″CD-ROM″))

  Returns ″false″
- IF(ISUPPER (″Map Designer″))

  Returns ″false″

Related functions

• ISALPHA	• LEAVEALPHANUM
• ISLOWER	• LEAVENUM
• ISNUMBER	• LEAVEPRINT
• LEAVEALPHA	

MEMBER

Use MEMBER when you need to know whether an object occurs within a series.

The MEMBER function searches a series, looking for a single specified object in the series. If any object in the series matches the specified object, MEMBER returns ″true″. If there is no match, MEMBER returns ″false″.

Syntax:

MEMBER (single-object-expression , series-object-expression)

MEMBER (single-object-expression , { literal, literal ... })

Meaning:

MEMBER (object_to_look_for , series_of_objects_to_look_at)

Returns:

″True″ or ″false″

MEMBER returns ″true″ if *object_to_look_for* matches one of the values in *series_of_objects_to_look_at*.

It returns ″false″ if *object_to_look_for* does *not* match at least one of the values in *series_of_objects_to_look_at*.

The two arguments, *object_to_look_for* and *series_of_objects_to_look_at*, must be objects of the same item interpretation or the same group type. For example, if *object_to_look_for* is a date/time item, *series_of_objects_to_look_at* must be a series of date/time items.

Examples

- MEMBER (EntityIDCode:Name, {″BT″ , ″ST″})

  This example tests whether **EntityIDCode** has one of a particular set of literal values.
- MEMBER (Store# , EntityIDCode:Name)

  This example tests whether **Store#** has the same value as any **EntityIDCode:Name**.

Related functions

- EXTRACT
- LOOKUP

NOT

Use the NOT function to test a condition and have it return the inverse of its ″true″ or ″false″ result. For example, you want the function to return ″true″ if the condition results in ″false″.

Syntax:

NOT (single-condition-expression)

Meaning:

NOT (condition_to_evaluate)

Returns:

″True″ or ″false″

NOT returns ″true″ if the condition evaluates to ″false″ and returns ″false″ if the condition evaluates to ″true″.

Examples

- NOT (Qty:.:InputFile = 0)

  This example returns ″false″ if the **Qty** equals 0 (the condition is true) and returns ″true″ if **Qty** does not equal 0 (the condition is false).
- IF (NOT (PRESENT (StartDate)), ″Unknown″, ″none″)

  This example returns ″unknown″ if **StartDate** is not present and returns ″none″ if **StartDate** is absent.

  Another way to test that an object is not present is to use the ABSENT function.

OFFSET

Use the OFFSET function when you need to know the position of a particular data object within its card object.

OFFSET returns an integer representing the offset of the specified object within the data.

Syntax:
OFFSET (single-object-expression)

Meaning:
OFFSET (object_whose_offset_is_needed)

Returns:
A single integer

OFFSET returns the offset, in bytes, of the specified object within its card object, beginning at offset 0. For an object that has an initiator, the offset will apply to the first byte of the data. For an object that is right-justified with pad characters, OFFSET will return the offset of the first byte of data.

The OFFSET function works the same for output objects as it does for input objects.

Examples

- OFFSET (Application:LoanData)

  In this example, if the first character of the first occurrence of the object **Application** occurs 210 bytes from offset 0 within **LoanData**, the OFFSET function returns 210.

Related function

- SIZE

OR

Use the OR function to test whether one of a series of conditions is true.

OR evaluates a series of conditions and returns ″true″ if at least one evaluates to ″true″; otherwise returns ″false″.

Syntax:
OR (series-condition-expression)

Meaning:
OR (conditions_to_evaluate)

Returns:
″True″ or ″false″

OR evaluates to ″true″ if *any* member of the argument evaluates to ″true″ and evaluates to ″false″ if *all* members of the argument evaluate to ″false″.

Examples

- Order(s)=IF (OR (Store:Table = Store#:Order:Input), Order:Input)

This example produces an **Order** if the **Store#** of an **Order** in Input matches any **Store** in **Table**.

Related function

- ALL

PARTITION

The PARTITION function checks to see if an occurrence of an object belongs to a certain partition. If the object is that partition, ″true″ is returned. Otherwise, ″false″ is returned.

Syntax:

PARTITION (single-object-expression, single-simple-object-name)

Meaning:

PARTITION (partitioned_object, simple_name_of_partition_to_check_for)

Returns:

″True″ or ″false″

PARTITION returns ″true″ if the data object of *partitioned_object* belongs to the partition represented by *simple_name_of_partition_to_check_for*. Otherwise, PARTITION returns ″false″.

Examples

- Assume that **Transaction** has been partitioned into three partitioned subtypes: **Invoice**, **Order**, and **Remittance**, as shown.

  The following rule could be used to detect whether a given **Transaction** is an **Invoice**:

  PARTITION (Transaction::Batch, Invoice)

  If the **Transaction** is an **Invoice**, PARTITION returns ″true″. If the **Transaction** is not an **Invoice** (for example, it is an **Order** or a **Remittance**), PARTITION returns ″false″ for that **Transaction**.

Related function

- GETPARTITIONNAME

PRESENT

The PRESENT function tests for the presence of an object.

PRESENT is commonly used with the IF function in a map rule to provide conditional logic. For example, if the object is present, do this; otherwise, do something else.

Similarly, PRESENT is commonly used with the WHEN function in component rules to provide conditional validation logic.

Syntax:

PRESENT (single-object-expression)

Meaning:

PRESENT (object_to_look_for)

Returns:
"True" or "false"

PRESENT returns "true" if the input argument does *not* evaluate to "none"; the object is present. It returns "false" if the input argument evaluates to "none"; the object is not present.

Examples

- PRESENT (Trailer:File)

  This example returns "true" if **Trailer** is present and returns "false" if **Trailer** is absent.
- IF (PRESENT(MiddleInitial::Input), MiddleInitial::Input, "***")

  This example in a map rule maps **MiddleInitial** if it is present. If **MiddleInitial** is not present, three asterisks are mapped.
- WHEN (PRESENT (AreaCode Field), PRESENT (PhoneNo Field))

  In this example, PRESENT is being used in conjunction with the WHEN function in a component rule to determine whether a particular object is valid.

Related functions

- ABSENT
- IF
- WHEN

SIZE

The SIZE function returns an integer representing the size of a specified object, exclusive of any pad characters.

Syntax:
SIZE (single-object-expression)

Meaning:
SIZE (object_whose_size_is_needed)

Returns:
A single integer

The SIZE function returns the size, in bytes, of *object_whose_size_is_needed*. The size returned does not include any pad characters that may be in the object.

The size of a group is the size, beginning with the first character of the first component and ending with the last character of the last component, of the group. If the group has delimiters, the infix delimiters are included in the size.

Examples

- SIZE (Transaction)

  Returns 8000 if the size of **Transaction** (without pad characters) is 8000 bytes

TESTOFF

The TESTOFF function tests a specified bit in a binary number item to see whether it is off.

Syntax:

TESTOFF (single-binary-number-expression , single-integer-expression)

Meaning:
TESTOFF (binary_number_to_test , bit_to_test)

Returns:
"True" or "false"

The value of *bit_to_test* specifies which bit of *binary_number_to_test* should be tested for the value 0. If *bit_to_test* has the value 1, it refers to the leftmost bit of *binary_number_to_test*.

The TESTOFF function returns "true" if the specified bit is off and returns "false" if the specified bit is on.

If *bit_to_test* is less than one or greater than the number of bits of *binary_number_to_test*, TESTOFF returns "false".

Examples

- TESTOFF (A, 16)

  Assume **A** is the two-byte binary value of "1", which is all zeros except for bit **16**. The binary representation of the value in **A** is 0001.

  This example returns "false".
- TESTOFF (A, 20)

  Returns "false" because bit **20** does not exist.

TESTON

The TESTON function tests a specified bit in a binary number to see if it is on.

Syntax:

TESTON (single-binary-number-expression, single-integer-expression)

Meaning:
TESTON (binary_number_to_test ,bit_to_test)

Returns:
"True" or "false"

The value of *bit_to_test* specifies the bit of *binary_number_to_test* to test for the value 1. If *bit_to_test* has the value 1, it refers to the leftmost bit of *binary_number_to_test*.

The function returns "true" if the specified bit is on; it has the value 1. It returns "false" if the specified bit is off; it has the value 0.

If *bit_to_test* is less than one or greater than the number of bits of *binary_number_to_test*, TESTON returns "false".

Examples

- TESTON (A , 16)

  Assume **A** is the two-byte binary value of ″1″, which is all zeros except for bit **16**. The binary representation of the value in **A** is 0001.

  This example returns ″true″.
- TESTON (A , 20)

  Returns ″false″ because bit **20** does not exist in a two-byte value.

VALID

You can use the VALID function to perform conditional processing based on whether an external interface function executes successfully.

VALID returns the result of the first argument if it is valid; otherwise, returns the second argument.

Syntax:

VALID (single-text-expressions , single-general-expression)

Meaning:

VALID (function_that_can_fail , return_value_if_function_fails)

Returns:

A single text expression

VALID returns the result of the evaluation of *function_that_can_fail* if it is valid. If the function fails, VALID returns *return_value_if_function_fails*.

The following functions can fail:

• DBLOOKUP	• GET
• DBQUERY	• PUT
• DDEQUERY	• RUN
• EXIT	

Examples

- SomeObject = VALID (RUN (″mymap.mmc″ , ″-OF1 mydata.txt″) , FAIL (″My RUN failed!″))

  If the RUN function returns an error return code, the VALID functions returns ″none″, the map aborts, and the message ″My RUN failed!″ is reported under ″Execution Summary″ in the execution audit log.

Related functions

- DBLOOKUP
- DBQUERY
- DDEQUERY

Chapter 12. Logical functions

ALL

The ALL function evaluates a series of conditions and returns ″true″ if they all evaluate to ″true″; otherwise returns ″false″.

Syntax:
ALL (series-condition-expression)

Meaning:
ALL (conditions_to_evaluate)

Returns:
True or false

The ALL function evaluates to ″true″ if *all* members of the input argument evaluate to ″true″.; it evaluates to ″false″ if *any* member of the input argument is ″false″.

Examples

- You can use ALL when you want to test whether all conditions of a series are true.
- PO (s)=IF (ALL(PO#:Line:Order = PO#:Line:Order[1]), Order, ″none″)

  If each and every **PO#** matches the **PO#** of the first **Order**, ALL evaluates to ″true″. Otherwise, ALL evaluates to ″false″.

Related functions

- NOT
- OR

EITHER

The EITHER function returns the result of the first argument that does not evaluate to ″none″.

Syntax:
EITHER (single-general-expression { , single-general-expression })

Meaning:
EITHER (try_this { , if_none_try_this })

Returns:
A single object

The EITHER return methods work in the following ways:

Returns...
When...

try_this
try_this does not evaluate to ″none″

if_none_try_this
try_this evaluates to ″none″

the next *if_none_try_this*
the first *if_none_try_this* evaluates to ″none″

Examples

- EITHER (OrderDate Field , CURRENTDATE ( ))

  If **OrderDate Field** does not evaluate to ″none″, it is returned. If **OrderDate Field** evaluates to ″none″, the current system date (using CURRENTDATE) is returned.
- EITHER (LOOKUP (PriorityCd:Msg , CustID:Msg = ″93X″) , 8)

  This example returns the result of the LOOKUP if that result does not evaluate to ″none″; otherwise, it returns the second argument of 8.
- EITHER (IF (SomeCode = ″C″ , CustomerID:Input) ,
  IF (SomeCode = ″S″ , SupplierID:Input) ,
  IF (SomeCode = ″O″ , Reference#:Input) ,
  ″####″)

  If **SomeCode** is **C** and **CustomerID:Input** has a value, EITHER returns the value of **CustomerID**. Otherwise, if **SomeCode** is **S** and **CustomerID:Input** has a value, EITHER returns the value of **SupplierID**. Otherwise, if **SomeCode** is **O** and **Reference#:Input** has a value, EITHER returns the value of **Reference#**. If none of those conditions result in a value, EITHER returns ″####″.
- You can use EITHER when you want a default value when an expression evaluates to ″none″ and the expression may cause common arguments to produce an unintended result. For example, use:

  EITHER (LOOKUP (PriorityCd:Msg , CustID:Msg = ″93X″) , 8)

  -instead of-

  IF (PRESENT (LOOKUP (PriorityCd:Msg , CustID:Msg = ″93X″)),
  LOOKUP (PriorityCd:Msg , CustID:Msg = ″93X″) , 8)

IF

You can use the IF function for conditional logic. For example, to return one of two objects depending on the evaluation of a condition.

IF evaluates a conditional expression, returning one value if true, another if false.

Syntax:

IF (single-condition-expression , single-general-expression
[, single-general-expression])

Meaning:
IF (test_this , result_if_true [, result_if_false])

Returns:
A single tem or single group

If *test_this* evaluates to ″true″, the result is *result_if_true*.

Otherwise, if *test_this* evaluates to ″false″, the result is *result_if_false*. If no *result_if_false* is specified, this evaluates to ″none″.

The *result_if_true* and *result_if_false* arguments must correspond to the same item interpretation or the same group type or either can be ″none″. For example, if

result_if_true evaluates to a number, *result_if_false* must also evaluate to a number. If *result_if_true* evaluates to a **LineItem** group, *result_if_false* must also evaluate to a **LineItem** group.

Examples

- IF (Quantity:LineItem:PO > 500, ″PRIORITY″, ″REGULAR″)

  This example tests the **Quantity** value. If that **Quantity** is > 500, the IF function evaluates to the value PRIORITY. If that **Quantity** is <= 500, the IF function evaluates to the value, REGULAR.
- IF (Status:Order = ″Special″, ″SPCL″)

  This example tests the **Status** value. If the **Status** is Special, the IF function evaluates to the value, SPCL. Otherwise, it evaluates to ″none″. Notice that this rule is equivalent to

  IF (Status:Order = ″Special″, ″SPCL″, ″none″)

ISALPHA

You can use ISALPHA when you need to know whether a text string is all alphabetic characters.

Syntax:
ISALPHA (single-text-expression)

Meaning:
ISALPHA (text_to_test)

Returns:
This function evaluates to a Boolean ″true″ or ″false″ and should only be used as a conditional expression within a logical function.

The ISALPHA function tests a text object to see if it is contains all alphabetic characters.

If *text_to_test* contains only alphabetic characters (for example, A-Z and a-z), ISALPHA evaluates to ″true″.

If *text_to_test* contains other than just alphabetic characters, ISALPHA returns ″false″.

Examples

- IF(ISALPHA (″AnywhereUSA″))

  Returns ″true″
- IF(ISALPHA (″Anywhere USA″))

  Returns ″false″
- IF(ISALPHA (″Mr. Brown″))

  Returns ″false″

Related functions

• ISLOWER	• LEAVEALPHANUM
• ISNUMBER	• LEAVENUM
• ISUPPER	• LEAVEPRINT

• LEAVEALPHA	

ISLOWER

The ISLOWER function tests a text object to see if it contains all lowercase alphabetic characters.

Syntax:
ISLOWER (series-text-expression)

Meaning:
ISLOWER (text_to_test)

Returns:
This function evaluates to a Boolean ″true″ or ″false″ and should only be used as a conditional expression within a logical function.

If *text_to_test* contains only lowercase alphabetic characters (for example, a-z), ISLOWER evaluates to ″true″.

If *text_to_test* contains other than lowercase alphabetic characters, ISLOWER returns ″false″.

Examples

- IF(ISLOWER (″company″))
  Returns ″true″
- IF(ISLOWER (″pots and pans″))
  Returns ″false″
- IF(ISLOWER (″Andrew″))
  Returns ″false″

Related functions

• ISALPHA	• LEAVEALPHANUM
• ISNUMBER	• LEAVENUM
• ISUPPER	• LEAVEPRINT
• LEAVEALPHA	

ISNUMBER

The ISNUMBER function tests a text object to determine whether it contains all numeric characters.

Syntax:
ISNUMBER (single-text-expression)

Meaning:
ISNUMBER (text_to_test)

Returns:
This function evaluates to a Boolean ″true″ or ″false″ and should only be used as a conditional expression within a logical function.

If *text_to_test* contains only digits (for example, 0-9), ISNUMBER evaluates to ″true″.

If *text_to_test* contains other than digits, ISNUMBER evaluates to ″false″.

Examples

- IF(ISNUMBER(″1″)
  Returns ″true″

Related functions

• ISALPHA	• LEAVEALPHANUM
• ISLOWER	• LEAVENUM
• ISUPPER	• LEAVEPRINT
• LEAVEALPHA	

ISUPPER

The ISUPPER function tests a text object to determine whether it contains all uppercase alphabetic characters.

Syntax:
ISUPPER (series-text-expression)

Meaning:
ISUPPER (text_to_test)

Returns:
This function evaluates to a Boolean ″true″ or ″false″ and should only be used as a conditional expression within a logical function.

If *text_to_test* contains only uppercase alphabetic characters (for example, A-Z), ISUPPER evaluates to ″true″.

If *text_to_test* contains other than uppercase alphabetic characters, ISUPPER returns ″false″.

Examples

- IF(ISUPPER (″BOMBAY″))
  Returns ″true″
- IF(ISUPPER (″CD-ROM″))
  Returns ″false″
- IF(ISUPPER (″Map Designer″))
  Returns ″false″

Related functions

• ISALPHA	• LEAVEALPHANUM
• ISLOWER	• LEAVENUM
• ISNUMBER	• LEAVEPRINT

• LEAVEALPHA	

NOT

Use the NOT function to test a condition and have it return the inverse of its ″true″ or ″false″ result. For example, you want the function to return ″true″ if the condition results in ″false″.

Syntax:
NOT (single-condition-expression)

Meaning:
NOT (condition_to_evaluate)

Returns:
″True″ or ″false″

NOT returns ″true″ if the condition evaluates to ″false″ and returns ″false″ if the condition evaluates to ″true″.

Examples

- NOT (Qty:.:InputFile = 0)

  This example returns ″false″ if the **Qty** equals 0 (the condition is true) and returns ″true″ if **Qty** does not equal 0 (the condition is false).
- IF (NOT (PRESENT (StartDate)), ″Unknown″, ″none″)

  This example returns ″unknown″ if **StartDate** is not present and returns ″none″ if **StartDate** is absent.

  Another way to test that an object is not present is to use the ABSENT function.

OR

Use the OR function to test whether one of a series of conditions is true.

OR evaluates a series of conditions and returns ″true″ if at least one evaluates to ″true″; otherwise returns ″false″.

Syntax:
OR (series-condition-expression)

Meaning:
OR (conditions_to_evaluate)

Returns:
″True″ or ″false″

OR evaluates to ″true″ if *any* member of the argument evaluates to ″true″ and evaluates to ″false″ if *all* members of the argument evaluate to ″false″.

Examples

- Order(s)=IF (OR (Store:Table = Store#:Order:Input), Order:Input)

  This example produces an **Order** if the **Store#** of an **Order** in Input matches any **Store** in **Table**.

Related function

- ALL

WHEN

You can use the WHEN function in a component rule to test for the validity of one object based on another object.

WHEN evaluates a condition. Then, based on that evaluation, evaluates another condition and returns ″true″ or ″false″.

Syntax:

WHEN (single-condition-expression , single-condition-expression [, single-condition-expression])

Meaning:

WHEN (condition1 , condition2 [, condition3])

Returns:

″True″ or ″false″

The following statements summarize how the WHEN function works:

- Returns ″true″ if *condition1* and *condition2* both evaluate to ″true″.
- Returns ″true″ if *condition1* evaluates to ″false″ and there is no *condition3* or if *condition1* evaluates to ″false″ and *condition3* evaluates to ″true″.
- Returns ″false″ if *condition1* evaluates to ″true″ and *condition2* evaluates to ″false″.
- Returns ″false″ if both *condition1* and *condition3* evaluate to ″false″.

Evaluation of the three arguments:

Condition 1	Condition 2	Condition 3	WHEN returns:
True	True		True
False			True
False		True	True
True	False		False
False		False	False

Examples

- WHEN (ABSENT(CatalogueField), ABSENT (QuantityField))
  Returns ″true″ when **CatalogueField** and **QuantityField** are both absent
- WHEN (PRESENT(ShipDate), PRESENT(InStock), PRESENT(BackOrderDate))
  Returns ″false″ when **ShipDate** and **BackOrderDate** are both absent

Chapter 13. Implementing a library EXIT function

You can develop a function in a library to be executed from an EXIT function.

EXIT function's library interface

Library functions are used within an EXIT function using information contained in the EXITPARAM structure. This method provides great flexibility for data passed to and from a map. For example, the map can pass binary data containing nulls, and there is no limitation on the length of the returned data. The method also allows functions to report additional information by providing a return code and error message.

Using the EXITPARAM Structure

Function prototype

The function to be executed must be a function in a library with the following prototype:

```
void MyFunc(LPEXITPARAM lpep);
```

Definition of the EXITPARAM structure

The definition of the EXITPARAM structure is as follows:

```
struct tagExitParamStruct
{
   DWORD         dwSize;
   DWORD         dwToLen;
   DWORD         dwFromLen;
   DWORD         dwMapInstance;
   void FAR *    lpv;
   LPSTR         lpszCmdLine;
   BYTE HUGE *   lpDataToApp;
   BYTE HUGE *   lpDataFromApp;
   UINT          uRetryCount;
   UINT          uRetryInterval;
   BOOL          bRollback;
   BOOL          bCleanup;
   int           nReturn;
   char          szErrMsg[100];
   char          szFile[260];
   void FAR *    lpMapHandle;
   void FAR *    lpInternal;
   void FAR *    lpCmdStruct;
   void FAR *    lpAdaptParms;
   void FAR *    lpContext;
   void FAR *    lpWildcard;
   void FAR *    lpfnMS;
   void FAR *    lpMS;
   DWORD         dwWildcardSize;
   LPSTR         lpszMapDirectory;
   WORD          wCardNum;
   WORD          wCleanupAction;
   WORD          wScope;
   UINT          uUnitSize;
   BOOL          bBurst;
   BOOL          bFromRule;
```

```
   BOOL          bSource;
   DWORD         dwRecords;
};
typedef struct tagExitParamStruct EXITPARAM;
typedef struct tagExitParamStruct FAR * LPEXITPARAM;
```

The engine environment sets up the `EXITPARAM` structure and the function should fill in the result. The engine will allocate and free the memory associated with **lpDataToApp**. The function must allocate the memory for **lpDataFromApp**. For all Windows platforms, use the Windows macro **GlobalAllocPtr** defined in **windowsx.h**. For all other platforms, use the `C-runtime malloc` function. The engine will free this memory.

Table 3. Components of EXITPARAM as used with the EXIT function.

Component	Used as	Usage
dwSize	Input	Size (in bytes) of EXITPARAM to assure correct compatibility
dwToLen	Input	Length (size in bytes) of lpDataToApp
dwFromLen	Output	Length (size in bytes) of lpDataFromApp
dwMapInstance		Not used
lpv		Not used
LpszCmdLine		Not used
lpDataToApp	Input	Data sent to the function. This is the third parameter of the EXIT function
lpDataFromApp	Output	Data sent back to the server from the function
uRetryCount		Not used
uRetryInterval		Not used
bRollback		Not Used
bCleanup		Not used
nReturn	Output	Return code based on the outcome of the function
szErrMsg	Output	String message based on nReturn
szFile		Not used
lpInternal		Not used
lpCmdStruct		Not used
lpAdaptParms		Not used
lpContext		Not used
lpWildcard		Not used
dwWildcardSize		Not used
lpszMapDirectory		Not used
wCardNum		Not used
wCleanupAction		Not used
wScope		Not used
uUnitSize		Not used
bBurst		Not used
lpfnMS		Not used.

Table 3. Components of EXITPARAM as used with the EXIT function. (continued)

Component	Used as	Usage
lpMS		Not used
dwRecords		Not used

There is no interaction with the **nReturn** or **szErrMsg** fields. However, this might change in a future release.

Chapter 14. Lookup and reference functions

CHOOSE

The CHOOSE function returns the object within a series whose position in the series corresponds to a specified number.

Syntax:

CHOOSE (series-object-name , single-integer-expression)

Meaning:

CHOOSE (from_these_objects , pick_the_nth_one)

Returns:

A single object whose index within the from_these_objects series matches the number specified by pick_the_nth_one. If that member of the series does not exist, CHOOSE returns ″none″.

You can use CHOOSE to use a variable value to specify the index for a particular object from a series.

Examples

- CHOOSE (Row:DBSelect , 2)

  Returns the second **Row**
- CHOOSE (Set:Claim , INDEX (Row:Header))

  Returns the **Set** within the **Claim** that corresponds to the index of the **Row** of **Header**.

Related function

- LOOKUP

DBLOOKUP

The DBLOOKUP function executes an SQL statement against a database. The SQL statement can be any permitted by your database management system or ODBC driver.

When the DBLOOKUP function is used in a map, the default **OnSuccess** action is adapter-specific. The default **OnFailure** action is to rollback any changes made during map processing. The default **Scope** will be integral unless the map is defined to run in bursts (which is the case when one or more inputs have the **FetchAs** property set to **Burst**).

There are two ways to specify arguments for DBLOOKUP.

You can use DBLOOKUP to execute an SQL statement when you want to execute a SELECT statement to retrieve a specific column value in a large table in a database using the value of another input, rather than defining the entire table as an input card and using the LOOKUP, SEARCHDOWN, or SEARCHUP functions.

You can use DBLOOKUP to execute an SQL statement when you want to execute a SELECT statement to retrieve a specific column value from a table or database that

might vary based on a parameter file. Using Meaning 2 of the DBLOOKUP function allows these parameters to be dynamically specified at run time.

Syntax:

DBLOOKUP (single-text-expression , single-text-expression , [single-text-literal])

Meaning:

1. DBLOOKUP (SQL_statement , mdq_filename, database_name)
2. DBLOOKUP (SQL_statement , parameters)

Returns:

A single text item

The DBLOOKUP function returns the results of the query in the same format as a query specified for a map input card, except that it does not include the last carriage return/linefeed. Because this information is removed, it is easier to make use of a single value extracted from a database.

Arguments for meaning 1

DBLOOKUP (SQL_statement , mdq_filename , database_name)

- **SQL_statement**

  The first argument is an SQL statement as a text string. This can be any valid SQL statement permitted by your database management system and supported by your database-specific driver. In addition to a fixed SQL statement, this argument can be a concatenation of text literals and data objects, enabling the concatenation of data values into your SQL statement.

- **mdq_filename**

  The second argument is the name of a database query file (**.mdq**) produced by the Database Interface Designer. It contains the definition of the database that the SQL statement is to be executed against. If the **.mdq** file is in a directory other than the directory of the map, the path must be specified.

  Note: The **.mdq** file is accessed at map build time and is not needed at runtime.

- **database_name**

  The third argument is the name of a database in the database query file (**.mdq**) as defined in the Database Interface Designer.

  If used in this way, both the **.mdq** filename and database name must be literals.

Arguments for meaning 2

DBLOOKUP (SQL_statement , parameters)

- **SQL_statement**

  The first argument is an SQL statement as a text string. This can be any valid SQL statement permitted by your database management system and supported by your database-specific driver. In addition to a fixed SQL statement, this argument can be a concatenation of text literals and data objects, enabling the concatenation of data values into your SQL statement.

- **parameters**

  The second argument is a set of parameters, either:

  – -MDQ *mdqfilename* -DBNAME *dbname*

  -or-

– -DBTYPE *database_type* [*database specific parameters*]

The keyword -MDQ is followed by the name of the database query file (**.mdq**) produced by the Database Interface Designer. This **.mdq** file contains the definition of the database. If the **.mdq** file is in a directory other than the directory of the map, the path must be specified. The **.mdq** filename is followed by the keyword -DBNAME and the database name as specified in the Database Interface Designer.

Using this syntax, the **.mdq** file is accessed at runtime and must be present.

The keyword -DBTYPE is followed by a keyword specifying the database type (for example, ODBC or ORACLE) followed, optionally, by database-specific parameters.

This syntax does not use an **.mdq** file, because the database-specific parameters provide the information required to connect to the database. Refer to the Resource Adapters documentation for detailed information on the database-specific parameters that can be specified.

When used with Meaning 2, DBLOOKUP must conform to these rules:

- All keywords (for example, -DBTYPE) can be upper or lowercase, but not mixed.
- A space is required between the keyword and its value (for example, -DBTYPE ODBC).
- The order of the keywords is not important.

  All database-specific parameters are optional.

Examples

Assume that you have a table named ″PARTS″ that contains the following data:

PART_NUMBER	PART_NAME
1	1/4″ x 3″ Bolt
2	1/4″ x 4″ Bolt

Assume that this database has been defined using the Database Interface Designer in a file named **mytest.mdq** and that the name of the database, as specified in the **.mdq** file, is **PartsDB**.

- DBLOOKUP (″SELECT PART_NAME from PARTS where PART_NUMBER =1″,
  ″mytest.mdq″,
  ″PartsDB″)

  Returns: ¼″ x 3″ Bolt

Using Meaning 2, you can specify the DBLOOKUP this way:

- DBLOOKUP(″SELECT PART_NAME from PARTS where PART_NUMBER =1″,
  ″-MDQ mytest.mdq -DBNAME PartsDB″)

  where both the **.mdq** file name and database name is specified.

Using Meaning 2, you can also specify the database type and the appropriate database-specific parameters:

- DBLOOKUP(″SELECT PART_NAME from PARTS where PART_NUMBER =1″ ,
  ″-DBTYPE ORACLE -CONNECT MyDB -USER janes ″)

Related functions

- DBQUERY
- EXTRACT
- FAIL
- LASTERRORCODE
- LASTERRORMSG
- LOOKUP
- SEARCHDOWN
- SEARCHUP
- VALID

For more examples using the DBLOOKUP function, see the Database Interface Designer documentation.

DBQUERY

The DBQUERY function executes an SQL statement against a database. The SQL statement can be any permitted by your database management system or ODBC driver.

When the DBQUERY function is used in a map, the default **OnSuccess** action is adapter-specific. The default **OnFailure** action is to rollback any changes made during map processing. The default **Scope** will be integral unless the map is defined to run in bursts (which is the case when one or more inputs have the **FetchAs** property set to **Burst**).

There are two ways to specify the arguments for DBQUERY. You can use DBQUERY [*Meaning 1*] to execute an SQL statement when you want to look up information in a database using a parameterized query that is based on another value in your data. If your SQL statement is a SELECT statement, the DBQUERY function may be used in conjunction with the RUN function to issue dynamic SELECT statements whose results can be used as input to another map.

You can also use the DBQUERY function [*Meaning 2*] to execute an SQL statement when the database, table, or other database parameters might vary; perhaps being supplied by a parameter file.

Syntax:

```
DBQUERY (single-text-expression , single-text-expression ,
 [ single-text-literal ] )
```

Meaning:

1. DBQUERY (SQL_statement , mdq_filename, database_name)
2. DBQUERY (SQL_statement , parameters)

Returns:

A single text item

If your SQL statement is a SELECT statement, the results of the query in the same format as a query specified as a map input card, including row delimiters and terminators, and so on.

If your SQL statement is anything other than a SELECT statement, ″none″.

Arguments for meaning 1

DBQUERY (SQL_statement, mdq_filename , database_name)

- **SQL_statement**

  The first argument is an SQL statement as a text string. This can be any valid SQL statement that is permitted by your database management system and supported by your database-specific driver. In addition to a fixed SQL statement, this argument can be a concatenation of text literals and data objects, enabling the concatenation of data values into your SQL statement.

- **mdq_filename**

  The second argument is the name of a database query file (**.mdq**) produced by the Database Interface Designer. It contains the definition of the database that the SQL statement is to be executed against. If the **.mdq** file is in a directory other than the directory of the map, the path must be specified.

  Note: The **.mdq** file is accessed at map build time and is not needed at runtime.

- **database_name**

  The third argument is the name of a database in the database query file (**.mdq**) as defined in the Database Interface Designer.

  If used in this way, both the **.mdq** filename and database name must be literals.

Arguments for meaning 2

DBQUERY (SQL_statement , parameters)

- The first argument is an SQL statement as a text string. This can be any valid SQL statement that is permitted by your database management system and supported by your database-specific driver. In addition to a fixed SQL statement, this argument can be a concatenation of text literals and data objects, enabling the concatenation of data values into your SQL statement.
- The second argument is a set of parameters, either:
  - -MDQ mdqfilename -DBNAME dbname

    -or-

  - -DBTYPE database_type [database specific parameters]

    The keyword -MDQ is followed by the name of the database query file (**.mdq**) produced by the Database Interface Designer. This **.mdq** file contains the definition of the database. If the **.mdq** file is in a directory other than the directory of the map, the path must be specified. The **.mdq** filename is followed by the keyword -DBNAME and the database name as specified in the Database Interface Designer.

    Note: Using this syntax, the **.mdq** file is accessed at runtime and must be present.

    The keyword -DBTYPE is followed by a keyword specifying the database type (for example, ODBC or ORACLE) followed, optionally, by database-specific parameters.

    Note: This syntax does not use an **.mdq** file, because the database-specific parameters provide the information required to connect to the database. Refer to the appropriate database adapter documentation for detailed information about database-specific parameters.

When used with Meaning 2, DBQUERY must conform to these rules:

- All keywords (for example, -DBTYPE) can be upper or lower case, but not mixed.
- A space is required between the keyword and its value (for example, -DBTYPE ODBC).
- The order of the keywords is not important.

  All database-specific parameters are optional.

Examples

Assume that you have a table named ″PARTS″ that contains the following data:

PART_NUMBER	PART_NAME
1	1/4″ x 3″ Bolt
2	1/4″ x 4″ Bolt

Also assume that this database has been defined using the Database Interface Designer in a file named **mytest.mdq** and that the name of the database, as specified in the **.mdq** file, is **PartsDB**.

```
DBQUERY ( "SELECT * from PARTS" ,  "mytest.mdq" ,  "PartsDB" )
```

Returns 1|¼″ x 3″ Bolt<cr><lf>2|¼″ x 4″ Bolt<cr><lf>

where <cr><lf> is a carriage return followed by a line feed.

Using Meaning 2, you can also specify the DBQUERY this way:

```
DBQUERY ( "SELECT * from PARTS" ,  "-MDQ mytest.mdq -DBNAME PartsDB" )
```

where both the **.mdq** file name and database name are specified.

Or, specify it this way, using Meaning 2 by specifying the database type and the appropriate database-specific parameters:

```
DBQUERY ( "SELECT * from PARTS" ,  "-DBTYPE ORACLE -CONNECT MyDB -USER
janes" )
```

Assume that you have an input file containing one order record. To map that order to another proprietary format, you also have a parts table with pricing information for every part for every customer, a very large table. Rather than using the entire parts table as the input to your map, you might use the RUN function with a DBQUERY to dynamically select only those rows from the parts table corresponding to the customer in the order file, as follows:

```
RUN ( "MapOrder.MMC" ,
  "IE2" + DBQUERY ( "SELECT * FROM Parts WHERE CustID = "
 + CustomerNo:OrderRecord:OrderFile + " ORDER BY PartNo" ,
"PartsDB.MDQ", "PartsDatabase" ) )
```

Related functions

- DBLOOKUP
- EXTRACT
- FAIL
- LASTERRORCODE

- LASTERRORMSG
- LOOKUP
- SEARCHUP
- SEARCHDOWN
- VALID

DDEQUERY

The DDEQUERY function allows you to interface to other Windows applications such as Trading Partner PC, Excel, and so forth, provided that certain criteria are met. For example, if you receive an Excel spreadsheet file, you must have the appropriate version of the Excel application installed (that is compatible with the file received) and the application must be open.

Syntax:

DDEQUERY (single-text-expression , single-text-expression, single-text-expression)

Meaning:

DDEQUERY (application_name , topic , text)

Returns:

A single text item from an application

Examples

- DDEQUERY (″excel″ , ″[MKTPRICE.XLS]Sheet1″, ″R8C1:R14C3″)

  In this example, DDEQUERY is used to get data from an Excel spreadsheet. The third argument, **R8C1:R14C3**, specifies the location of the data in the spreadsheet. (In Excel, the 8th row, 1st column to the 14th row, 3rd column is A8:C14.) The content of this spreadsheet range is returned as a single text item.

  This example assumes that the application, the map, and the spreadsheet all reside in the same directory. If they are not in the same directory you must add the path. For example:

  DDEQUERY (″excel″ , ″c:\spreadsheet[MKTPRICE.XLS]Sheet1″ , ″R8C1:R14C3″)
- DDEQYUERY (″tppc″,″PartnerX″,″BGyourEDIode″)

  In this example, DDEQUERY is used as a request to Trading Partner PC.

Related functions

• EXIT	• LASTERRORMSG
• FAIL	• PUT
• GET	• RUN
• LASTERRORCODE	• VALID

EXTRACT

Use EXTRACT whenever you need only particular members of a series returned-those that meet a certain condition. An example may be only POs that contain backordered items.

The EXTRACT function can only be used in a map rule. It cannot be used in a component rule.

The EXTRACT function returns all members of a series for which a specified condition is true.

Syntax:

EXTRACT (series-object-expression, single-condition-expression)

Meaning:

EXTRACT (objects_to_extract, condition_to_evaluate)

Returns:

A series object.

The result is each member of *series_to_search* for which the condition specified by *condition_to_evaluate* evaluates to ″true″. EXTRACT returns ″none″, if no member of *series_to_search* has a corresponding *condition_to_evaluate* that evaluates to ″true″.

Examples

- EXTRACT (PO:Transaction , Store# = Location:PO:Transaction)

  This example returns all **PO**s, individually, whose **Location** is a particular **Store#**.
- EXTRACT (Row:DBSelect , ProcessFlag Column:Row:DBSelect = ″Y″)

  This example returns all **Rows** that have a **ProcessFlag Column** value of ″Y″.

Related Functions

- CHOOSE
- LOOKUP
- SEARCHDOWN
- SEARCHUP

GETANDSET

You can use GETANDSET when you have an input file that keeps track of control information that serves as an input to your map and the control information in the file needs to be updated based on processing that occurs in your map.

The GETANDSET function gets a fixed length value from the input data stream, updates that value in the input data stream, and returns either the original or updated value.

Syntax:

GETANDSET (single-fixed-size-item-object-name, single-item-expression, single-integer-expression)

Meaning:

GETANDSET (original_value ,new_value, integer_that_determines_which_value_to_return)

Returns:

A single-fixed-size-item and replaces the original value in the input data stream.

This function does not support decrementing integers past 0 into negative numbers.

GETANDSET updates an input by finding the object represented by *original_value* and replacing it with the value represented by the *new_value*. The function returns either the *original_value* or the *new_value*, depending on the value of the *integer_that_determines_which_value_to_return*. For example,

- If *integer_that_determines_which_value_to_return* has the value 1, *original_value* is returned.
- If *integer_that_determines_which_value_to_return* has the value 2, *new_value* is returned.
- If *integer_that_determines_which_value_to_return* has any other value, *original_value* is returned.
- If one of the input arguments evaluates to ″none″, GETANDSET returns ″none″.

  The original value specified for *new_value* must be a fixed size item. During map execution, the original value is updated to reflect the evaluation of GETANDSET. When the source is a file, that file will be updated after map completion. However, if the source is a database, message, or application, the content of the source is in memory and the source, itself, is not updated.

Examples

- New Tracking# = GETANDSET (Tracking#:Card, Tracking#:Card + 3, 1)

  This rule finds the object **Tracking#:Card** (assume that it has the value 6), adds 3 to it, giving 9, and replaces the 6 with the 9 in the data location of **Tracking#:Card**. Subsequent references to **Tracking#:Card** will find its value to be 9. If **Card** is an input card whose source is a file, the file is rewritten with the new value.

  Because the third input argument (*integer_that_determines_which_value_to_return*) is 1, the returned value is the original value of 6.

GETDIRECTORY

The GETDIRECTORY function returns the full path (directory) for the compiled map file or the source or destination associated with a specified card object.

You can use GETDIRECTORY in a map rule or component rule when you need the full path (directory) of either the compiled map or of a data source or destination.

Syntax:

GETDIRECTORY ([single-simple-object-name])

Meaning:

GETDIRECTORY ([card_or_object_for_which_directory_is_needed])

Returns:

A single text item

Without an argument, GETDIRECTORY returns the full path associated with the compiled map.

With an argument, the following occurs:

- From a map rule, the function returns the full path of the source or destination that is associated with the card. In a map rule, the argument must be the name of the card for which to get the directory.

- From a component rule, the function returns the full path of the source or destination that is associated with the active source or destination.

  The GETDIRECTORY command without arguments will return the directory of the compiled map, regardless of whether it is used in a map rule or in a component rule.

Examples

- GETDIRECTORY (OrderFile)

  If the card **OrderFile** is associated with the data file, *install_dir***order.txt**, GETDIRECTORY returns *install_dir*\.

- GETDIRECTORY ( )

  If the compiled map on the HP-UX is **/maps/prod/mymap.mmc**, GETDIRECTORY returns **/maps/prod/**.

  Suppose you want to run the map, **MyMap**, from a component rule on **Record** in your input to determine whether input customer names are valid. **MyMap** has two inputs and one output. The first input is the customer name to up. The second input is a lookup file. The output is a text item whose value is ″valid″ or ″error″.

  The name of the lookup file is constant; it is always **XREF_TBL.TXT**. However, its location may vary; it will always be in the same directory as the data file used as the source you are trying to validate. For example, if the name of the data file used as the source is **C:\SHR\ABC\INPUT.TXT**, the lookup file name is **C:\SHR\ABC\XREF_TBL.TXT**. If the data file name is **/local/data/somefile**, the lookup file name is **/local/data/XREF_TBL.TXT**, and so forth.

  You could use this component rule on **Record** to determine whether the customer name is valid:

  RUN (″MyMap.mmc″ , ″-IE1S10″ + CustomerName:$ + ″ -IF2 ″ + GETDIRECTORY ( ) + ″XREF_TBL.TXT″ + ″ -OE1″) = ″VALID″

Related functions

- GETFILENAME
- GETRESOURCENAME

GETLOCALE

The GETLOCALE function returns the locale setting of the computer.

Syntax:

GETLOCALE ()

Meaning:

GETLOCALE ()

Returns:

A single text item

The GETLOCALE function returns the locale setting of the computer where the map runs. The locale is returned in the format specified by the operating system. Generally, the return format is an ISO Language Code (as defined by ISO-639) in combination with an ISO Country Code (as defined by ISO-3166) when applicable. The language codes are two lower-case letters and the country codes are two upper-case letters. For example, **en_US** is the code for English (United States).

This function has no arguments but requires parentheses.

Table 4. Examples

System locale	Returns
French	fr
Korean	ko
Brazilian Portuguese	pt_BR
Taiwanese Chinese	zh-TW

GETFILENAME

The GETFILENAME function returns the file name for a file adapter source or target of a specified card object. Without an argument, it returns the adapter command associated with the active source or destination.

You can use GETFILENAME in a map rule or component rule when you need the data file name.

Syntax:

GETFILENAME ([single-simple-object-name])

Meaning:

GETFILENAME ([card_for_which_resource_info_is_needed])

Returns:

A single text item

With an argument, GETFILENAME returns the file name for a file source or target of a specified card object. Without an argument, the function returns the source or destination name associated with the active source or destination.

Examples

- GETFILENAME (OrderFile)

  If the card **OrderFile** is associated with the data file, *install_dir***order.txt**, GETFILENAME returns *install_dir***order.txt**.

  Suppose you want to run the map **MyMap** from a component rule on **Record** in your input to determine if input customer names are valid. **MyMap** has two inputs and one output. The first input is the customer name to be looked up. The second input is a lookup file. The output is a text item whose value is ″valid″ or ″error″.

  The name of the lookup file may vary; it must correspond to the name of the data file used as the source you are trying to validate. For example, if the data file name is **c:\DATA.AAA**, the lookup file name is **c:\LOOKUP.AAA**. If the data file name is **c:\DATA.XYZ**, the lookup file name is **c:\LOOKUP.XYZ**, and so on.

  You could use this component rule on **Record** to determine whether the customer name is valid:

  RUN (″MyMap.mmc″ , ″-IE1S10″ + CustomerName:$ + ″ -IF2 LOOKUP.″ + RIGHT (GETFILENAME ( ) , 3) + ″ -OE1″) = ″VALID″

Related functions

- GETDIRECTORY
- GETRESOURCENAME

GETPARTITIONNAME

You can use GETPARTITIONNAME when you need to know the name of the partition to which the data for the given object belongs.

Syntax:
GETPARTITIONNAME (single-partitioned-object-expression)

Meaning:
GETPARTITIONNAME (partitioned_object)

Returns:
A single simple object name

GETPARTITIONNAME returns a text item that represents the name of the partition to which the data for *partitioned_object* belongs.

Examples

- GETPARTITIONNAME (Transaction:File)

  The type defined by **Transaction** is a partitioned object that has three partitions: **Add**, **Delete**, and **Modify**. In this example, if the input data for **Transaction** belongs to the **Delete** partition, the GETPARTITIONNAME function returns ″delete″.

Related functions

- PARTITION

GETRESOURCEALIAS

The GETRESOURCEALIAS function returns the resource alias value specified in a Resource Registry resource name file (.mrn).

Syntax:
GETRESOURCEALIAS (sinlge-text-expression, single-text-expression)

Meaning:
GETRESOURCEALIAS (single-text-expression, single-text-expression)

Returns:
A single text item

The GETRESOURCEALIAS function loads a Resource Registry resource configuration file and retrieves the specified alias value. The values are defined as part of the Global object in the .mrc file.

In the below example the resource configuration file is defined as:

```
<?xml version="1.0" encoding="UTF-8"?>
<ResourceCfg>

 <Global>
  <ResourceFile ActiveVirtualServer="test">Company.mrn</ResourceFile>
 </Global>
</ResourceCfg>
```

The resource name file is defined as:

```
<?xml version="1.0" encoding="UTF-8"?>
<MRN>
 <VirtualServerSet>
  <VirtualServer>test</VirtualServer>
```

```
 </VirtualServerSet>
 <Resource>
  <Name>company</Name>
  <Value Server="test" encrypt="OFF">IBM</Value>
 </Resource>
</MRN>
```

..when used as a part of this rule:

```
=resourcelib->GETRESOURCEALIAS("company.mrc","%company%")
```

In this scenario, the return value is ″IBM″.

When an absolute path is not defined in the location, the default location is the map directory.

GETRESOURCENAME

The GETRESOURCENAME function returns the adapter source or target command of a specified card object. Without an argument, it returns the adapter command associated with the active source or destination.

You can use GETRESOURCENAME in a component or map rule when you need to know the name of a source or destination.

Syntax:

GETRESOURCENAME ([single-simple-object-name])

Meaning:

GETRESOURCENAME ([card_for_which_resource_info_is_needed])

Returns:

A single text item

With an argument, the source or destination name that is associated with the card. Without an argument, returns the source or destination name associated with the active source or destination.

Examples

- GETRESOURCENAME (OrderFile)

  If the card **OrderFile** is associated with the data file, *install_dir***order.txt**, the GETRESOURCENAME function returns *install_dir***order.txt**.

Related functions

- GETDIRECTORY
- GETFILENAME

GETTXINSTALLDIRECTORY

The GETTXINSTALLDIRECTORY function returns the WebSphere Transformation Extender product installation directory.

Syntax:

GETTXINSTALLDIRECTORY ()

Meaning:

GETTXINSTALLDIRECTORY ()

Returns:
A single text item

GETTXINSTALLDIRECTORY returns the product installation directory. For example, a Windows operating system might return: **C:\Program Files\IBM\Websphere Transformation Extender 8.1**, or a UNIX operating system would return the DTX_HOME_DIR value in the setup script.

This function has no arguments but requires parentheses.

INDEX

You can use the INDEX function when you need to select or test particular objects based on their occurrence, or to add a sequence number to output objects.

INDEX returns an integer that represents the index of an object relative to its nearest contained object, counting only valid objects.

INDEX cannot be used in a component rule.

Syntax:
INDEX (single-object-name)

Meaning:
INDEX (object_for_which_to_get_index)

Returns:
A single integer

The result is the index of *object_for_which_to_get_index*.

- If *object_for_which_to_get_index* is an input, this will be the index within all valid objects.
- If *object_for_which_to_get_index* is an output, this will be the index within all objects (valid and invalid).
- Returns 0 if the input argument is ″none″.

  The difference between INDEXABS and INDEX is that INDEXABS counts both valid and invalid instances, whereas INDEX counts only valid instances.

Examples

- Message (s) = IF (INDEX (Message:Input) > 3, Message:Input, ″none″)

  For example, there are five **Messages** in **Input**. The first three evaluations of this rule return ″none″. The fourth evaluation returns **Message[4]**. The fifth evaluation returns **Message[5]**.
- Invoice (s) = MyMap (Invoice Segment:Input, INDEX ($))

  For the first evaluation of **MyMap**, INDEX($) is 1. For the second evaluation of **MyMap**, INDEX($) is 2.

Related functions

- CHOOSE
- COUNT
- COUNTABS
- INDEXABS

INDEXABS

You can use INDEXABS when you need the absolute occurrence of a particular object across all occurrences, rather than across only valid occurrences.

The INDEXABS function returns an integer that represents the index of an object relative to its nearest contained object, counting both valid and invalid instances of the object.

Syntax:

INDEXABS (single-object-name)

Meaning:

INDEXABS (object_for_which_to_get_index)

Returns:

A single integer

INDEXABS returns an integer that represents the absolute index of *object_for_which_to_get_index*. The integer indicates the instance this object that is in the set of *all* instances of the object, including both valid and invalid occurrences. Returns 0 if the input argument is ″none″.

- If *object_for_which_to_get_index* is an input, this will be the index within all members of the series, including valid objects, invalid objects, and existing ″none″s.
- If *object_for_which_to_get_index* is an output, this will be the index within all existing members of the series, including existing ″none″s.

  The difference between INDEXABS and INDEX is that INDEXABS counts both valid and invalid instances, as well as existing ″none″s, whereas INDEX counts only valid instances.

Examples

- INDEXABS (Message Record:Order:PO_File)

  For this example, that **Order** contains the following **Messages**:

  Message Record[1] Valid

  Message Record[2 Error

  Message Record[3] Valid

  In a map rule, INDEXABS (MessageRecord[3]:Order:PO_File) would evaluate to 3.

  If the INDEX function was used, INDEX (Message Record[3]:Order:PO) would evaluate to 2.

Related function

- INDEX

LASTERRORCODE

The LASTERRORCODE function returns a text item whose value is the last error code returned by one of a specified set of functions during map execution.

You can use LASTERRORCODE to interrogate or report the error code returned by one of the external interface functions.

Syntax:

LASTERRORCODE ( )

Meaning:

LASTERRORCODE ( )

Returns:

A single text item

LASTERRORCODE has no arguments but it requires parentheses.

LASTERRORCODE returns a text item whose value is the last error code returned by one of a specified set of functions during map execution.

The following functions can fail:

- DBLOOKUP
- DBQUERY
- DDEQUERY
- EXIT
- GET
- PUT
- RUN

Examples

- Message = VALID (RUN (″Map1Msg.mmc″ , ″-AE -OMMSMQ1B `-QN .\aqueue -CID 2001′″), FAIL (″Failure on RUN (″ + TEXT (LASTERRORCODE ( )) + ″):″ + LASTERRORMSG ( )))

  In this example, the LASTERRORCODE and LASTERRORMSG functions are being used in conjunction with the FAIL and VALID functions to fail (abort) the map if the map executed by the RUN function (**Map1Msg.mmc**) fails. In this example, the map fails and returns the error code and error message reported by the RUN function using the LASTERRORCODE and LASTERRORMSG functions.

  If **Map1Msg** fails because one or more of its inputs was invalid, **Message** is assigned a value of ″none″. The map aborts and the following message is reported in the execution audit log:

  Failure on RUN (8): One or more inputs was invalid.

Related functions

- DBLOOKUP
- DBQUERY
- DDEQUERY

LASTERRORMSG

The LASTERRORMSG function returns a text item whose value is the message corresponding to the last error code returned by one of a specified set of functions during map execution.

Syntax:

LASTERRORMSG ( )

Meaning:

LASTERRORMSG ( )

Returns:
A single text item

Although LASTERRORMSG has no arguments, it does require parentheses.

LASTERRORMSG returns a text item whose value is the message corresponding to the last error code returned by one of a specified set of functions during map execution.

The following is the list of functions that can fail:

• DBLOOKUP	• GET
• DBQUERY	• PUT
• DDEQUERY	• RUN
• EXIT	

Examples

- Message = VALID (RUN (″Map1Msg.mmc″ , ″-AE -OMMSMQ1B `-QN .\aqueue -CID 2001′″), FAIL (″Failure on RUN (″ + TEXT (LASTERRORCODE ( )) + ″):″ + LASTERRORMSG ( )))

  In this example, the LASTERRORCODE and LASTERRORMSG functions are being used in conjunction with the FAIL and VALID functions to fail (abort) the map if the map executed by the RUN function (**Map1Msg.mmc**) fails. In this example, the map fails and returns the error code and error message reported by the RUN function using the LASTERRORCODE and LASTERRORMSG functions.

  If **Map1Msg** fails because one or more of its inputs was invalid, Message is assigned a value of ″none″. The map aborts and the following message is reported in the execution audit log:

  Failure on RUN (8): One or more inputs was invalid.

Related functions

- DBLOOKUP
- DBQUERY
- DDEQUERY

LOOKUP

You can use LOOKUP to find an occurrence of an object that meets a certain condition.

The LOOKUP function sequentially searches a series, returning the first member of the series that meets a specified condition.

Syntax:
LOOKUP (series-object-expression , single-condition-expression)

Meaning:
LOOKUP (series_to_search , condition_to_evaluate)

Returns:
A single object

LOOKUP returns the first member of *series_to_search* for which *condition_to_evaluate* evaluates to ″true″; it returns ″none″ if no member of *series_to_search* meets the condition specified by *condition_to_evaluate*.

Examples

- LOOKUP (Account#:Customer , Company Name:Customer = ″ACME″)

  This example returns the **Account#** of **Customer** whose **Company Name** is ACME.
- LOOKUP (Part#:Row:DBSelect , Model#:Row:DBSelect = ModelCode:Legacy & Serial#:Row:DBSelect > ″123″)

  This example returns the **Part#** of **DBSelect** where the **Model#** in that row matches the **ModelCode** of **Legacy** and the **Serial#** is greater than 123.

Related functions

- CHOOSE
- EXTRACT

  Note: LOOKUP differs from EXTRACT in that LOOKUP returns the *first* member of *series_to_search* that meets the *condition_to_evaluate*, while EXTRACT returns all members (one at a time) of *series_to_search* that meet the *condition_to_evaluate*.
- SEARCHUP
- SEARCHDOWN

  Note: LOOKUP performs a sequential search over *series_to_search*. Use LOOKUP if *series_to_search* is *not* ordered. Using SEARCHUP when that series is in ASCII ascending order or SEARCHDOWN when that series is in ASCII descending order can save time if the series being searched is large.

MEMBER

Use MEMBER when you need to know whether an object occurs within a series.

The MEMBER function searches a series, looking for a single specified object in the series. If any object in the series matches the specified object, MEMBER returns ″true″. If there is no match, MEMBER returns ″false″.

Syntax:

MEMBER (single-object-expression , series-object-expression)

MEMBER (single-object-expression , { literal, literal ... })

Meaning:

MEMBER (object_to_look_for , series_of_objects_to_look_at)

Returns:

″True″ or ″false″

MEMBER returns ″true″ if *object_to_look_for* matches one of the values in *series_of_objects_to_look_at*.

It returns ″false″ if *object_to_look_for* does *not* match at least one of the values in *series_of_objects_to_look_at*.

The two arguments, *object_to_look_for* and *series_of_objects_to_look_at*, must be objects of the same item interpretation or the same group type. For example, if *object_to_look_for* is a date/time item, *series_of_objects_to_look_at* must be a series of date/time items.

Examples

- MEMBER (EntityIDCode:Name, {″BT″ , ″ST″})

  This example tests whether **EntityIDCode** has one of a particular set of literal values.
- MEMBER (Store# , EntityIDCode:Name)

  This example tests whether **Store#** has the same value as any **EntityIDCode:Name**.

Related functions

- EXTRACT
- LOOKUP

SEARCHDOWN

You can use the SEARCHDOWN function when data is sorted in ASCII, descending order and you need to look up data within the sorted data.

SEARCHDOWN performs a binary search on a series sorted in ASCII descending order, returning a related object that corresponds to the item found.

Syntax:

SEARCHDOWN (series-object-expression, series-item-object-expression , single-item-expression)

Meaning:

SEARCHDOWN (corresponding_object_to_return, descending_items_to_search , item_to_match)

Returns:

A single object

SEARCHDOWN performs a binary search on the item series of *descending_items_to_search*. The *descending_items_to_search* must be sorted in ASCII descending order. The value to search for is specified as the *item_to_match*. The object returned (*corresponding_object_to_return*) must be related to *descending_items_to_search* by a common object name.

If no match is found, SEARCHDOWN returns ″none″.

Examples

- SEARCHDOWN (Age Column:Row:DBSelect ,
  SSN Column:Row:DBSelect ,
  SSN_Value:Message)

  If there are ten rows in **DBSelect**, the search starts by comparing the first **SSN Column** of the fifth row with the **SSN_Value** in **Message**. If the result matches, SEARCHDOWN returns the first **Age Column** of that **Row**. If the value of **SSN Column** is less than the **SSN_Value** in **Message**, the search continues with the third **Row**. If the value of **SSN Column** is greater than the **SSN_Value** in **Message**, the search continues with the seventh Row in **DBSelect**. The search continues in this fashion until either a match is found or until one **Row** is

selected. If there is more than one **SSN Column** for the selected **Row**, a similar search is initiated for all **SSN Column**'s for the selected **Row** in **DBSelect**.

SEARCHDOWN returns the first **Age Column** for the selected **Row** of **DBSelect**.

Related functions

- EXTRACT
- LOOKUP
- SEARCHUP

SEARCHUP

Use SEARCHUP when data is sorted in ASCII ascending order and you need to look up data within the sorted data using the value on which the data is sorted.

The SEARCHUP function performs a binary search on a series sorted in ASCII ascending order, returning a related object that corresponds to the item found.

Syntax:

SEARCHUP (series-object-expression, series-item-object-expression, single-item-expression)

Meaning:

SEARCHUP (corresponding_object_to_return, ascending_items_to_search, item_to_match)

Returns:

A single object

SEARCHUP performs a binary search on the item series of *ascending_items_to_search*. The *ascending_items_to_search* must be sorted in ASCII ascending order. The value to search for is specified as the *item_to_match* and must be of the same type as the *ascending_item_to_search*. The object returned (*corresponding_object_to_return*) must be related to *ascending_items_to_search* by a common object name.

If no match is found, SEARCHUP returns "none".

Examples

- SEARCHUP (Age Column:Row:DBSelect, SSN Column:Row:DBSelect, SSN_Value:Message)

  If there are ten rows in **DBSelect**, the search starts by comparing the first **SSN Column** of the fifth **Row** with the **SSN_Value** in **Message**. If the result matches, SEARCHUP returns the first **Age Column** of that **Row**. If the value of **SSN Column** is greater than the **SSN_Value** in **Message**, the search continues with the third **Row**. If the value of **SSN Column** is less than the **SSN_Value** in **Message**, the search continues with the seventh **Row** in **DBSelect**. The search continues in this fashion until either a match is found or until one **Row** is selected. If there is more than one **SSN Column** for the selected **Row**, a similar search is initiated for all **SSN Column**'s for the selected **Row** in **DBSelect**.

  SEARCHUP returns the first **Age Column** for the selected **Row** of **DBSelect**.

Related functions

- EXTRACT
- LOOKUP

- SEARCHDOWN

SORTDOWN

You can use the SORTDOWN function to sort objects in a series in ASCII descending sequence. The function returns a series containing the values from the input series in ASCII descending order.

Syntax:
SORTDOWN (series-item-expression)

Meaning:
SORTDOWN (item_series_to_sort)

Returns:
A series object

Returns the values in *item_series_to_sort* in ASCII descending order.

Examples

- SORTDOWN (Abbr:File)

  In this example, if **Abbr** has these values:

  ABC, GHI, DEF

  SORTDOWN returns these values as a series as:

  GHI, DEF, ABC
- The following table displays the results of using the SORTDOWN and SORTUP functions for a typical series of text items.

Original Input Series	Result using SORTDOWN	Result using SORTUP
Clams Casino	shrimps	1 Shrimp
Grouper	raw oysters	22 Shrimp
groupers	oysters	A
Shrimp	lobster tails	A 1 A shrimp
shrimps	lobster	A1A Shrimp
lobster	groupers	AA
lobster tails	clams	AAAAA
oysters	aaaaa	Clams Casino
raw oysters	aa	Grouper
clams	a	Rock Lobster
SHRIMP	Snapper	SHRIMP
1 Shrimp	Snapper	Shark Fin Soup
22 Shrimp	Shrimp	Shrimp
A 1 A shrimp	Shark Fin Soup	Snapper
A1A Shrimp	SHRIMP	Snapper
AAAAA	Rock Lobster	a
aaaaa	Grouper	aa
aa	Clams Casino	aaaaa
AA	AAAAA	clams

Original Input Series	Result using SORTDOWN	Result using SORTUP
A	AA	groupers
a	A1A Shrimp	lobster
Rock Lobster	A 1 A shrimp	lobster tails
Snapper	A	oysters
Snapper	22 Shrimp	raw oysters
Shark Fin Soup	1 Shrimp	shrimps

Related functions

- SEARCHDOWN
- SEARCHUP
- SORTUP
- UNIQUE

SORTUP

Use SORTUP when you need to sort the objects of a series in ASCII ascending sequence. The SORTUP function returns a series containing the values from an input series in ASCII ascending order.

Syntax:

SORTUP (series-item-expression)

Meaning:

SORTUP (item_series_to_sort)

Returns:

A series object

SORTUP returns the values in *item_series_to_sort* in ASCII ascending order.

Examples

- SORTUP (Abbr:File)

  In this example, if **Abbr** had the values ABC, GHI, DEF, the SORTUP function would return these values as a series: ABC, DEF, GHI.

Related functions

- SEARCHDOWN
- SEARCHUP
- SORTDOWN
- UNIQUE

UNIQUE

The UNIQUE function returns a series containing all ″unique″ members of a series.

UNIQUE can only be used in a map rule, not in a component rule.

Syntax:

UNIQUE (series-object-expression)

Meaning:
UNIQUE (series_to_evaluate)

Returns:
A series object

UNIQUE evaluates to a series containing all unique members of *series_to_evaluate* and evaluates to ″none″ if *series_to_evaluate* evaluates to ″none″.

Examples

- UNIQUE (PartNumber:Inventory:File)
  Returns the unique **PartNumbers** in **Inventory:File**
- COUNT (UNIQUE (Customer:Order:File))
  Returns the number of unique **Customers** in **Order:File**

Related functions

- SORTDOWN
- SORTUP

Chapter 15. Math and statistics functions

ABS

The ABS function returns the absolute value of a number.

Syntax:

ABS (single-number-expression)

Meaning:

ABS (number)

Returns:

A single number; the absolute value of a number

Examples

- ABS (-3)
  Returns 3
- ABS (3)
  Returns 3
- AvailableCredit has a value of -69.42
  ABS (AvailableCredit) returns 69.42
- FlexDollars has a value of 50
  ABS ((100 - FlexDollars)/2) returns 25

ACOSINE

Use the ACOSINE function to calculate the arccosine of a value.

Syntax:

ACOSINE (single-number-expression)

Meaning:

ACOSINE (number_to_convert)

Returns:

A single number item

The result is the arccosine of the converted value.

ASIN

Use the ASIN function to calculate the arcsine of a value.

Syntax:

ASIN (single-number-expression)

Meaning:

ASIN (number_to_convert)

Returns:

A single number item

The result is the arcsine of the converted value.

ATAN

The ATAN function calculates the arctangent of a value.

Syntax:
ATAN (single-number-expression)

Meaning:
ATAN (number_to_convert)

Returns:
A single number item

ATAN2

The ATAN2 function calculates the arctangent of y/x.

Syntax:
ATAN2 (single-number-expression)

Meaning:
ATAN (number_to_convert)

Returns:
A single number item

COSINE

The COSINE function calculates the cosine of a value.

Syntax:
COSINE (single-number-expression)

Meaning:
COSINE (number_to_convert)

Returns:
A single number item

COSINEH

The COSINEH function calculates the hyperbolic cosine of a value.

Syntax:
COSINEH (single-number-expression)

Meaning:
COSINEH (number_to_convert)

Returns:
A single number item

COUNT

You can use the COUNT function to return an integer representing the number of valid input or output objects in a series.

Syntax:
COUNT (series-object-expression)

Meaning:
COUNT (valid_objects_to_count)

Returns:
A single integer

The result is the number of *valid_objects_to_count*. If the input argument evaluates to ″none″, COUNT returns 0.

COUNT does not count existing ″none″s unless its group was defined as an explicit format with a **Track** setting of Places.

Examples

- COUNT (Claim Record:Patient File)
  This example returns the number of valid **Claim Record** objects in **Patient File**.
- COUNT (Class IN Transcript)
  This example returns the number of valid **Class** objects in **Transcript**.
- COUNT (UNIQUE (Class IN Transcript))
  This example returns the number of valid **Unique Class** objects in **Transcript**.

Related functions

- COUNTABS

COUNTABS

You can use COUNTabs to count the input or output objects in a series, regardless of the validity of the object. an integer representing the number of input objects in a series.

Unlike COUNT, COUNTABS includes both valid and invalid objects in a series.

Syntax:
COUNTABS (series-object-expression)

Meaning:
COUNTABS (objects_to_count)

Returns:
A single-integer

The result is the number of *objects_to_count*. If the input argument evaluates to ″none″, COUNTABS returns 0.

COUNTABS does not count existing ″none's″ unless its group was defined as an explicit format with a **Track** setting of **Places**.

Examples

- COUNTABS (Claim Record:Patient File)
  This example returns the number of **Claim Record** objects in **Patient File**.
- COUNTABS (Class IN Transcript)
  This example returns the number of **Class** objects in **Transcript**.

Related functions

- COUNT

EXP

The EXP function calculates the exponential of a value.

Syntax:
EXP (single-number-expression)

Meaning:
EXP (number_to_convert)

Returns:
A single number item

FACTORIAL

The FACTORIAL function calculates the factorial of a value.

Syntax:
FACTORIAL (single-number-expression)

Meaning:
FACTORIAL (number_to_convert)

Returns:
A single number item

FROMBASETEN

You can use FROMBASETEN when you need to convert numbers to a base other than 10.

The FROMBASETEN function converts an integer to a text item that can be interpreted as a number, using positional notation of the base specified.

Syntax:
FROMBASETEN (single-integer-expression , single-integer-expression)

Meaning:
FROMBASETEN (positive_integer_to_convert , base_to_convert_to)

Returns:
A single text item

FROMBASETEN returns a text item that results from converting *positive_integer_to_convert* to a text item that can be interpreted as a number using positional notation of the base specified by *base_to_convert_to*.

If *base_to_convert_to* is less than 2 or greater than 36, FROMBASETEN evaluates to "none". Resulting text item characters A-Z are interpreted as digits having decimal values from 10-35, respectively. The characters returned are uppercase.

Example

- FROMBASETEN (18 , 2)
  Returns the value 10010
- FROMBASETEN (123 , 8)
  Returns the value 173

Related function

- TOBASETEN

INT

You can use the INT function when you need only the integer portion of a number.

Syntax:

INT (single-number-expression)

Meaning:

INT (number_to_convert)

Returns:

A single integer

INT returns the integer portion of a number. The result is the integer part of *number_to_convert*. Any fractional part after the decimal point is dropped.

Examples

- INT (1.45)

  Returns 1
- INT (3.6)

  Returns 3
- INT (Purchase:Amt - Discount:Amt)

  Subtracts **Discount:Amt** from **Purchase:Amt** and returns the result as a whole number.

Related functions

- MOD
- ROUND
- TRUNCATE

LOG

The LOG function calculates the logarithms of a value.

Syntax:

LOG (single-number-expression)

Meaning:

LOG (number_to_convert)

Returns:

A single number item

LOG10

The LOG10 function calculates the logarithms for base 10 of a value.

Syntax:

LOG10 (single-number-expression)

Meaning:

LOG10 (number_to_convert)

Returns:
A single number item

MAX

The MAX function returns the maximum value from a series of number, date, time, or text values.

Syntax:
MAX (series-item-expression)

Meaning:
MAX (series_of_which_to_find_max)

Returns:
A single number

The result is the maximum value in the input argument series: number, text, or date/time.

Examples

- MAX (UnitPrice:Input)
  If the values for **UnitPrice** are {20, 10, 100}, MAX returns 100.
- MAX(EXTRACT(DueDate:Book:Library, CheckedOut:Book:Library = ″Y″))
  Returns the maximum (latest) **DueDate** for a book that is checked out from the library.

Related functions

- MIN

MIN

Use MIN when you need the minimum value from a series of number, date, time, or text values.

The MIN function returns the minimum value from a series.

Syntax:
MIN (series-item-expression)

Meaning:
MIN (series_of_which_to_find_min)

Returns:
A single number

The result is the minimum value of the input series: number, text, or date/time.

Examples

- MIN (UnitPrice:Input)
  If the values for **UnitPrice** are {20,10,100}, MIN returns 10.
- MIN (StartTime:.:Schedule)
  Returns the minimum (earliest) **StartTime** in **Schedule**.

Related functions

- MAX

MOD

Use MOD when you need the modulus of an integer and a number.

The MOD function returns the modulus that remains after a number is divided by an integer.

Syntax:

MOD (single-number-expression , single-integer-expression)

Meaning:

MOD (dividend , divisor)

Returns:

A single integer

The result is the remainder (modulus) after *dividend* is divided by *divisor*. The result has the same sign as *divisor*.

The *dividend* is first divided by the integer *divisor*, resulting in a quotient. The modulus is calculated by multiplying the integer portion of the quotient by *divisor* and then subtracting that product from *dividend*.

If *divisor* is 0, MOD returns ″none″.

Examples

- MOD (3 , 2) or MOD (-3, 2)
  Returns 1
- MOD (3, -2) or MOD (-3, -2)
  Returns -1
- MOD (-3, 2) or MOD (-3, -2)
  Returns 1
- MOD (-3, -2) or MOD (-3, -2)
  Returns -1

POWER

The POWER function calculates x raised to the power of y.

Syntax:

POWER (single-number-expression, single-number-expression)

Meaning:

POWER (base_number, exponent_number)

Returns:

A single number item

RAND

The RAND function returns a pseudorandom number.

Syntax:
RAND ()

Meaning:
RAND ()

Returns:
A single number item

ROUND

The ROUND function rounds a number to a specified number of decimal places. If the number of decimal places is not specified, the number is rounded to a whole number. The result is in character number format.

Syntax:
ROUND (single-number-expression [, single-integer-expression])

Meaning:
ROUND (number_to_round [, number_of_decimal_places])

Returns:
A single number

ROUND converts *number_to_round* to character format, if necessary, and then produces the value of *number_to_round* rounded to the number of decimal places specified by *number_of_decimal_places*. If *number_of_decimal_places* is not specified, *number_to_round* is rounded to the nearest whole number.

Examples

- ROUND (1.46 , 1)
  Returns 1.5
- ROUND (1.46)
  Returns 1

Related functions

- TRUNCATE

SIN

The SIN function calculates the sine of a value.

Syntax:
SIN (single-number-expression)

Meaning:
SIN (number_to_convert)

Returns:
A single number item

SINH

The SINH function calculates the hyperbolic sine of a value.

Syntax:
SINH (single-number-expression)

Meaning:
SINH (number_to_convert)

Returns:
A single number item

SQRT

The SQRT function returns the square root of a number.

Syntax:
SQRT (single-number-expression)

Meaning:
SQRT (number)

Returns:
A single number

SQRT returns the square root of *number*.

Examples

- SQRT (4)
  Returns 2

SUM

The SUM function calculates the sum of a series of numbers.

Syntax:
SUM (series-number-expression)

Meaning:
SUM (series_to_sum)

Returns:
A single number

SUM returns the sum of all members in *series_to_sum*.

Examples

- SUM (Quantity:LineItem)
  This example calculates the sum of all the **Quantity** objects of **LineItem**.

TAN

The TAN function calculates the tangent of a value.

Syntax:
TAN (single-number-expression)

Meaning:
TAN (number_to_convert)

Returns:
A single number item

TANH

The TANH function calculates the hyperbolic tangent of a value.

Syntax:
TANH (single-number-expression)

Meaning:
TANH (number_to_convert)

Returns:
A single number item

TOBASETEN

See description in ″Conversion Functions″.

TRUNCATE

The TRUNCATE function removes decimal places from a number, leaving a specified number of decimal places.

You can use TRUNCATE with a second argument to truncate a number to a specified number of decimal places or without a second argument to reduce a number to an integer by removing all decimal places.

Syntax:
TRUNCATE (single-number-expression[, single-integer-expression])

Meaning:
TRUNCATE (number_to_truncate[, number_of_decimal_places])

Returns:
A single number

TRUNCATE first converts *number_to_truncate* to character format, if necessary. It then truncates that number by removing decimal places to the right of *number_of_decimal_places*. If *number_of_decimal_places* is not used, the number is truncated to an integer.

Examples

- TRUNCATE (3.9292 , 2)
  Returns 3.92
- TRUNCATE (3.9292)
  Returns 3

Related functions

- INT
- ROUND

Chapter 16. Text functions

BCDTOTEXT

The BCDTOTEXT function converts the digits in a BCD (Binary Coded Decimal) item to a text item containing the digits of the BCD-encoded item as a string of characters.

Syntax:

BCDTOTEXT (single-text-expression)

Meaning:

BCDTOTEXT (BCD_item_to_convert)

Returns:

A single text item

BCD_item_to_convert is converted from BCD format to a text string containing the digits of the BCD-encoded value as a string of characters.

Numbers in BCD format have two decimal digits in each byte. Each half-byte, therefore, can contain a binary value from 0000 (which represents the digit 0) through 1001, which represents the digit 9). Based on this definition, the following applies:

- If any half-byte of the BCD number contains the binary value 1101 or 1111, that half-byte is ignored.
- If the BCD item contains the binary value 1010, 1011, 1100, or 1110, the output of the function is ″none″.

Examples

- BCDTOTEXT (Qty:Item)

  If **Qty** is x`1234', the result is 1234.
- BCDTOTEXT (DiscountAmt)

  If **DiscountAmt** is x`0123', the result is 0123.
- BCDTOTEXT (TotalDollars)

  If **Total** is x`F123', the result is 123.

Related functions

- BCDTOHEX
- BCDTOINT
- TEXTTOBCD

COUNTSTRING

You can use the COUNTSTRING function when you need to know the number of times a specific text string appears within another text string. The function begins to look for the character string from the first position of the first string, and proceeds forward one byte at a time.

Syntax:

COUNTSTRING (single-text-expression , single-text-expression)

Meaning:
COUNTSTRING (text_to_search , text_to_find_and_count)

Returns:
A single-integer

COUNTSTRING returns an integer that represents the number of times that a specified character string appears in another character string.

The result is a number representing the number of times *text_to_find_and_count* appears within t*ext_to_search*. If either *text_to_search* or *text_to_find_and_count* evaluates to ″none″, COUNTSTRING returns 0.

Examples

- COUNTSTRING (″banana″ , ″a″)
  Returns a value of 3
- COUNTSTRING (″aaaa″ , ″aa″)
  Returns a value of 3.

Related functions

- FIND
- LEFT
- MID
- RIGHT

CPACKAGE

CPACKAGE specifies the character set of the output of the function. From that point onward, the data is treated as if it were in that character set. If the data is not in the specified character set, you get the wrong answer.

The character set is required to be specified in this function. If you choose not to specify a character set, you should use the original version of the PACKAGE function.

Syntax:
CPACKAGE (single-object-expression , ″character-set-of-object-content″)

Meaning:
CPACKAGE (object_to_convert, object_character_set)

Returns:
A single text item

The second argument, *object_character_set*, represents the character set of the resulting object. Character set codes are listed in ″Character set codes″.

Examples

In this example, the group **Record** has an initiator of ″#″, a terminator of ″@″ and a delimiter of ″,″ with the following data:

″#1339X10A,491.38,Green,42x54@″

- CPACKAGE (Record:Card, ″ASCII″)
  Returns: #1339X10A,491.38,Green,42x54@

CSERIESTOTEXT

CSERIESTOTEXT specifies the character set of the output of the function. From that point onward, the data is treated as if it were in that character set. If the data is not in the specified character set, you get the wrong answer.

The character set is required to be specified in this function. If you choose not to specify a character set, you should use the original version of the SERIESTOTEXT function.

Syntax:

CSERIESTOTEXT (series-object-expression , ″character-set-of-object-content″)

Meaning:

CSERIESTOTEXT (series_to_convert, object_character_set)

Returns:

A single text item

The *series_to_convert* argument concatenates the series of the input argument, including nested delimiters but excluding initiators and terminators.

The second argument, *object_character_set*, represents the character set of the resulting object.

Examples

In this example, you have the following data that represents bowler information for a bowling league:

Andrews, Jessica:980206:JBC:145:138:177:159

Little, Randy:980116:BBK:175:168

Wayne, Richard:980102:JBC:185:204:179:164:212

Each record consists of the bowler's name, the date of their last game played, a team code and one or more bowling scores. **Record** is defined as a group that is infix delimited by a colon.

Using the following rule produces results of the concatenation of all scores for all of the bowlers, even though the scores are not all contiguous within the data.

- = CSERIESTOTEXT (Score Field:Bowler:Input, ″ASCII″)

  Returns: 145138177159175168185204179164212

You can change the rule to concatenate the list of scores to the bowler's name using the following rule:

- = BowlerName Field:Bowler:Input + ″ ->″ +
  CSERIESTOTEXT (Score Field:Bowler:Input, ″ASCII″)

  Returns:

  Andrews, Jessica -> 145138177159

  Little, Randy -> 175168

  Wayne, Richard -> 185204179164212

In this example, you have an input number that is of variable size, followed by a name. There is no syntax that separates the number from the name. You can define the number as a group with **Byte(s)** as a component and provide a component rule for **Byte(s)**, such as:

ISNUMBER ($)

Based on this, the number can be distinguished from the name. When mapping, collect all the bytes of the number back again. You can use CSERIESTOTEXT to do this.

CTEXT

CTEXT specifies the character set of the output of the function. From that point onward, the data is treated as if it were in that character set. If the data is not in the specified character set, you get the wrong answer.

The character set is required to be specified in this function. If you choose not to specify a character set, you should use the original version of the TEXT function.

Syntax:

CTEXT (single-object-expression , "character-set-of-object-content")

Meaning:

CTEXT (object_to_convert , object_character_set)

Returns:

A single text item

The first argument, *object_to_convert*, represents the object that is converted to a text item, excluding the initiator and terminator of the input object.

The second argument, *object_character_set*, represents the character set of the resulting object.

Example

In this example, the group **Record** has an initiator of the pound sign (**#**), a terminator of the at sign (**@**), and a delimiter of a comma (**,**), and uses the following data:

#1339X10A,491.38,Green,42x54@

- CTEXT (Record:card, "ASCII")
  Returns: 1339X10A,491.38,Green,42x54

The initiator and terminator are not included because only the content of the object is converted to text.

DATETOTEXT

The DATETOTEXT function converts a date object or expression to a text item.

Syntax:

DATETOTEXT (single-date-expression)

Meaning:
DATETOTEXT (date_to_convert)

Returns:
A single text item

If *date_to_convert* is a date object name, this returns the date as a text item formatted according to the presentation of the date object.

If *date_to_convert* is a date expression produced by a function, this returns the date as a text item formatted according to the presentation of the output argument of that function.

Examples

- DATETOTEXT (ShipDate)

  In this example, **ShipDate** is converted from a date to text. If **ShipDate** has a CCYYMMDD presentation, the resulting text item will have that presentation, as well.
- DATETOTEXT (CURRENTDATETIME ("{MM/DD/CCYY}"))

  In this example, CURRENTDATETIME evaluates and returns a date in MM/DD/CCYY format. Then DATETOTEXT evaluates and returns a text string that is that date in MM/DD/CCYY format.

  For example, use DATETOTEXT, to do text concatenation. The FROMDATETIME function provides greater flexibility in specifying the format of the resulting text item.

Related Functions

- FROMDATETIME
- NUMBERTOTEXT
- TEXT
- TEXTTODATE
- TEXTTONUMBER
- TEXTTOTIME
- TIMETOTEXT
- TODATETIME

FILLLEFT

The FILLLEFT function returns a text item of the length specified. The output is filled on its left with the specified pad value.

You can use FILLLEFT when you have a value that needs to be of a fixed size with a variable number of leading characters with a specified value.

Syntax:
FILLLEFT (single-text-expression , single-text-expression, single-integer-expression)

Meaning:
FILLLEFT (text_to_fill , pad_character, pad_to_length)

Returns:
A single text item

The FILLLEFT function returns the text string that results from padding out *text_to_fill* on its left side with the *pad_character* up to *pad_to_length* bytes.

If the pad-length argument is less than the number of bytes in the text to fill, no padding will appear.

Examples

- FILLLEFT (AcctID:Transaction , ″0″, 5)

  If **AcctID** has the value 14, FILLLEFT returns 00014
- FILLLEFT (NUMBERTOTEXT (InvoiceAmt) , ″*″ , 10)

  If **InvoiceAmt** has the value 24.75, FILLLEFT returns *****24.75

Related functions

• FILLRIGHT	• SQUEEZE
• LEAVEALPHA	• SUBSTITUTE
• LEAVEALPHANUM	• TRIMLEFT
• LEAVENUM	• TRIMRIGHT
• LEAVEPRINT	

FILLRIGHT

The FILLRIGHT function returns a text item of the length specified. The output is filled on its right with the specified pad value.

You can use FILLRIGHT when you have a value that needs to be of a fixed size with a variable number of trailing characters of a specified value.

Syntax:

FILLRIGHT (single-text-expression , single-text-expression , single-integer-expression)

Meaning:

FILLRIGHT (text_to_fill , pad_character, pad_to_length)

Returns:

A single text item

FILLRIGHT returns the text string that results from padding out *text_to_fill* on its right side with the *pad_character* up to *pad_to_length* bytes.

If the pad-length argument is less than the number of bytes in the text to fill, no padding will appear.

Examples

- FILLRIGHT (LastName:Contact, ″ ″ , 25)

  If **LastName** has the value Peterson, FILLRIGHT returns Peterson followed by 17 spaces.

Related functions

• FILLLEFT	• SQUEEZE

• LEAVEALPHA	• SUBSTITUTE
• LEAVEALPHANUM	• TRIMLEFT
• LEAVENUM	• TRIMRIGHT
• LEAVEPRINT	

FIND

The FIND function looks for one text string within another text string and returns to its starting position, if found.

Syntax:

FIND (single-text-expression , single-text-expression
[, single-number-expression])

Meaning:
FIND (text_to_find, where_to_look[, position_to_start_the_search])

Returns:
A single integer

FIND returns the starting position of the text item specified by *text_to_find* within the text item specified by *where_to_look*. A third argument (*position_to_start_the_search*) can be used to specify the location in *where_to_look* for the FIND to begin. Bytes in the text are numbered from left to right, with the leftmost byte being position 1.

If *text_to_find* is ″none″, FIND evaluates to ″none″.

If a third argument is not used or *position_to_start_the_search* evaluates to a negative number, it is assumed to be 1. If *position_to_start_the_search* evaluates to a number greater than the size of *where_to_look*, FIND evaluates to ″none″.

If *text_to_find* is not found in the *where_to_look* string, FIND evaluates to 0.

Examples

- FIND (″id″, ″Florida″)
  Returns the value 5
- FIND (″id″, ″Florida″, 8)
  Returns 0 because the 8 (*position_to_start_the_search*) is greater than the size of *where_to_look*
- FIND (″\″, ″mypath″,2)
  Returns 0 because the string ″\″ was not found in argument 2

Related functions

- LEFT
- MID
- RIGHT

HEXTEXTTOSTREAM

HEXTEXTTOSTREAM is the reverse of STREAMTOHEXTEXT. You can use the HEXTEXTTOSTREAM function to assign a binary text value to a character text item represented by hexadecimal pairs.

HEXTEXTTOSTREAM returns a binary text stream whose value is the evaluation of input character text represented by hexadecimal pairs.

Syntax:

HEXTEXTTOSTREAM (single-text-expression)

Meaning:

HEXTEXTTOSTREAM (series_of_hex_pairs)

Returns:

A single byte stream item

This function returns a binary text stream item whose value is the evaluation of input character text in *series_of_hex_pairs*, ignoring <WSP> characters between the hexadecimal pairs. White space characters include space, horizontal tab, carriage return, and line feed characters.

Input formats

The following table shows an example of input in its character text representation as viewed through the character editor, and in its ASCII code representation (binary text stream) as viewed through the hex editor. Each pair of binary text in the hex view represents one character in the character view of the character text.

Input (″41 42 43 44″)	Editor View	Value
Character text (hex pairs)	Character	″41 42 43 44″
ASCII code representation (binary text stream)	Hex	0x3431203432203433203434

Examples

- HEXTEXTTOSTREAM (″41 42 43 44″)

  Returns the evaluated value of the input (ASCII) character text string ″41 42 43 44″ as the output (ASCII) character text string ″ABCD″ as viewed in the character editor. (The hex view of the input is 0x3431203432203433203434. The hex view of the output is 0x41424344.)

- HEXTEXTTOSTREAM (″0D 0A 00″)

  Returns the evaluated value of the input (ASCII) character text string ″0D 0A 00″ as the output (ASCII) character text string ″<CR><LF><NULL>″ as viewed in the character editor. (The hex view of the input is 0x3044203041203030. The hex view of the output is 0x0D0A00.)

  See Design Studio Introduction documentation for a list of special symbols.

Related functions

- SYMBOL
- STREAMTOHEXTEXT

LEAVEALPHA

The LEAVEALPHA function removes all non-alphabetic characters from a specified text item.

You can use LEAVEALPHA to remove non-alphabetic characters such as symbols or numbers from a text item.

Syntax:
LEAVEALPHA (single-text-expression)

Meaning:
LEAVEALPHA (text_to_change)

Returns:
A single text item

LEAVEALPHA returns a string containing only the alphabetic characters (for example, A-Z and a-z) in *text_to_change*.

Examples

- LEAVEALPHA (″A-b-C-1$3″)
  Returns: AbC

Related functions

- ISALPHA
- ISLOWER
- ISNUMBER
- ISUPPER
- LEAVEALPHANUM
- LEAVENUM
- LEAVEPRINT

LEAVEALPHANUM

The LEAVEALPHANUM function removes all non-alphanumeric characters (such as symbols) from a specified text item.

Syntax:
LEAVEALPHANUM (single-text-expression)

Meaning:
LEAVEALPHANUM (text_to_change)

Returns:
A single text item

LEAVEALPHANUM returns a string containing only the alphanumeric characters (for example, A-Z, a-z and 0-9) in *text_to_change*.

Examples

- LEAVEALPHANUM (″A-b-C-1$3″)
  Returns: AbC13

Related functions

• ISALPHA	• LEAVEALPHA
• ISLOWER	• LEAVENUM
• ISNUMBER	• LEAVEPRINT
• ISUPPER	

LEAVENUM

The LEAVENUM function removes all non-numeric characters from a text item. For example, you can use LEAVENUM when you want to remove all alphabetic characters and symbols from a text string.

Syntax:
LEAVENUM (single-text-expression)

Meaning:
LEAVENUM (text_to_change)

Returns:
A single text item

LEAVENUM returns a string containing only the numeric characters (for example, 0-9) in *text_to_change*.

Examples

- LEAVENUM (″A-b-C-1$3″)
  Returns: 13

Related functions

• ISALPHA	• LEAVEALPHA
• ISLOWER	• LEAVEALPHANUM
• ISNUMBER	• LEAVEPRINT
• ISUPPER	

LEAVEPRINT

The LEAVEPRINT function removes all non-printable characters from a text item.

Syntax:
LEAVEPRINT (single-text-expression)

Meaning:
LEAVEPRINT (text_to_change)

Returns:
A single text item

LEAVEPRINT returns a string containing only printable characters in *text_to_change*.

Examples

- LEAVEPRINT ("A-b<SP>C-1$3<CR><LF>")
  Returns: A-b C-1$3

Related functions

• ISALPHA	• LEAVEALPHA
• ISLOWER	• LEAVEALPHANUM
• ISNUMBER	• LEAVENUM
• ISUPPER	

LEFT

The LEFT function returns a specified number of characters from a text expression beginning with the leftmost byte of a text item.

You can use LEFT when you need a specific part of a text item. For example, a customer number might have several uses and sometimes only the first 10 characters are needed. Therefore, LEFT can be used to return only the leftmost 10 characters.

Syntax:
LEFT (single-text-expression , single-integer-expression)

Meaning:
LEFT (text_to_extract_from , number_of_characters_to_extract)

Returns:
A single text item

LEFT returns the leftmost *number_of_characters_to_extract* characters from *text_to_extract_from* starting at the first (the leftmost) character in *text_to_extract_from*.

If *number_of_characters_to_extract* evaluates to an integer whose value is less than 1, LEFT evaluates to "none". If *number_of_characters_to_extract* evaluates to an integer whose value is greater than the size of *text_to_extract_from*, LEFT evaluates to the entire value of *text_to_extract_from*.

Examples

- LEFT ("Abcd", 2)
  Returns Ab
- LEFT ("Abcd", 6)
  Returns Abcd
- LEFT (LastName + ", " + FirstName, 25)
  Returns the leftmost 25 characters of the text string resulting from the concatenation of **LastName** and **FirstName** (separated by a comma and a space).

Related functions

- RIGHT
- MID

- FIND

LOWERCASE

The LOWERCASE function converts an alphabetic text item to all lowercase characters.

Syntax:
LOWERCASE (single-text-expression)

Meaning:
LOWERCASE (text_to_convert)

Returns:
A single text item

LOWERCASE produces a text item in which each byte from the input has been converted to lowercase. Any numeric or symbol characters in the text item remain unchanged.

Examples

- LOWERCASE (″A1b2C!″)
  Returns: a1b2c!

Related functions

- ISLOWER
- ISUPPER
- UPPERCASE

MAX

The MAX function returns the maximum value from a series of number, date, time, or text values.

Syntax:
MAX (series-item-expression)

Meaning:
MAX (series_of_which_to_find_max)

Returns:
A single number

The result is the maximum value in the input argument series: number, text, or date/time.

Examples

- MAX (UnitPrice:Input)
  If the values for **UnitPrice** are {20, 10, 100}, MAX returns 100.
- MAX(EXTRACT(DueDate:Book:Library, CheckedOut:Book:Library = ″Y″))
  Returns the maximum (latest) **DueDate** for a book that is checked out from the library.

Related functions

- MIN

MID

You can use the MID function when you need specific characters from a text item. MID returns one or more characters from a text item.

Syntax:

MID (single-text-expression, single-number-expression, single-number-expression)

Meaning:
MID (source_text, position_to_start_the_search, number_of_characters)

Returns:
A single text item

MID extracts one or more characters from *source_text* where *position_to_start_the_search* is the position of the first character to extract and *number_of_characters* specifies the number of characters to extract. The first character (leftmost) of *source_text* has a starting position of 1.

If *position_to_start_the_search* is greater than the length of *source_text*, MID returns "none". If *position_to_start_the_search* or *number_of_characters* is less than one, MID returns "none". If the rightmost number of characters of *source_text*, starting at *position_to_start_the_search*, is less than *number_of_characters*, MID returns the rightmost characters starting at the position specified in *position_to_start_the_search*.

Examples

- MID ("abc123", 5 , 3)

  Returns 23
- MID ("abc123", 7 , 1)

  Returns "none" because argument2 is larger than the number of characters in argument1
- MID ("abc123", -1 , 3)

  Returns "none" because argument2 is a negative number
- MID ("abc123", 2 , -2)

  Returns "none" because argument3 is a negative number

Related functions

- FIND
- LEFT
- RIGHT

MIN

Use MIN when you need the minimum value from a series of number, date, time, or text values.

The MIN function returns the minimum value from a series.

Syntax:
MIN (series-item-expression)

Meaning:
MIN (series_of_which_to_find_min)

Returns:
A single number

The result is the minimum value of the input series: number, text, or date/time.

Examples

- MIN (UnitPrice:Input)
  If the values for **UnitPrice** are {20,10,100}, MIN returns 10.
- MIN (StartTime:.:Schedule)
  Returns the minimum (earliest) **StartTime** in **Schedule**.

Related functions

- MAX

NUMBERTOTEXT

The NUMBERTOTEXT function converts a character number to a text item that looks like the original object.

You can use NUMBERTOTEXT when you need an object that is defined as a number converted to an object defined as text. This is useful when you need to concatenate text, however, the FROMNUMBER function provides greater flexibility in specifying the format of the resulting text item.

Syntax:
NUMBERTOTEXT (single-number-expression)

Meaning:
NUMBERTOTEXT (number_to_convert)

Returns:
A single text item

The resulting text looks like the input argument. The result is truncated, if necessary.

Examples

- NUMBERTOTEXT (ROUND (1000 - 24.75, 3))
  This example converts the result of the calculation (rounded to 3 decimal places) to text, resulting in 975.250.
- NUMBERTOTEXT (PurchaseNumber)
  This example converts **PurchaseNumber** from a number to text.

Related functions

- FROMNUMBER
- TEXTTONUMBER
- TODATETIME
- TONUMBER

PACKAGE

The PACKAGE function converts a group or item object to a text item, including its initiator, terminator, and any delimiters it contains.

Syntax:
PACKAGE (single-object-expression)

Meaning:
PACKAGE (object_to_convert)

Returns:
A single text item

The PACKAGE function converts *object_to_convert*, which must be a type reference to a text item, including the type reference's initiator, terminator, and all delimiters. PACKAGE differs from TEXT in that it includes the initiator and terminator of the specified type reference.

Examples

- PACKAGE (Record:Card)

  Returns: #1339X10A,491.38,Green,42x54@

  For this example, the group **Record** has an initiator of ″**#**″, a terminator of ″**@**″ and a delimiter of ″**,**″. The data looks like this: ″#1339X10A,491.38,Green,42x54@″.

Related functions

- DATETOTEXT
- NUMBERTOTEXT
- SERIESTOTEXT
- TIMETOTEXT
- TEXT

  PACKAGE differs from TEXT because it includes the initiator and terminator of the input object.

PROPERCASE

Use PROPERCASE to convert the first alphabetic character in each word to uppercase and all remaining characters to lowercase.

Syntax:
PROPERCASE (single-text-item-expression)

Meaning:
PROPERCASE (text_item_to_convert)

Returns:
A single text item

The PROPERCASE function views the text string as containing a series of ″words″ where the delimiter between words is the space character. For each ″word″ in *text_item_to_convert*, PROPERCASE converts the first alphabetic character (for example, A-Z and a-z) found to uppercase and all other characters are converted to lowercase.

Examples

- PROPERCASE (″sallY jo BRADLEY″)
  Returns: Sally Jo Bradley
- PROPERCASE (″One AND only 1one.″)
  Returns: One And Only 1One

Related functions

- ISALPHA
- ISLOWER
- ISUPPER
- LOWERCASE
- UPPERCASE

REVERSEBYTE

Use the REVERSEBYTE function when you need the bytes in the opposite sequence. REVERSEBYTE reverses the byte order of an item.

Syntax:
REVERSEBYTE (single-item-expression)

Meaning:
REVERSEBYTE (item_to_reverse)

Returns:
A single item

This function reverses the byte order of *item_to_reverse*.

Examples

- REVERSEBYTE (″HI MOM!″)
  Returns ″!MOM IH″
- RIGHT (FullName, FIND (″ ″ , REVERSEBYTE (FullName)) - 1)
  If **FullName** is ″Alyce N. Wunderland″, the above example uses REVERSEBYTE to reverse the characters in **FullName** (resulting in ″dnalrednuW .N ecylA″). Then, the FIND function is evaluated to locate the first space in the resultant string (between the ″W″ and the ″.″) that would result in a value of 11. Finally, the RIGHT function is evaluated to take the rightmost 10 (11-1) characters of **FullName**; providing the final result of ″Wunderland″.

RIGHT

You can use RIGHT when you need a specific part of a text item. For example, a customer number may have several uses and sometimes only the last three characters are needed. RIGHT can be used to return only the rightmost three characters.

The RIGHT function returns a specified number of characters from a text expression beginning with the rightmost byte of a text item.

Syntax:
RIGHT (single-text-expression , single-integer-expression)

Meaning:
RIGHT (text_to_extract_from , number_of_characters_to_extract)

Returns:
A single text item

RIGHT returns the rightmost *number_of_characters_to_extract* characters from *text_to_extract_from* starting at the last (the rightmost) character in *text_to_extract_from*.

If *number_of_characters_to_extract* evaluates to an integer whose value is less than 1, RIGHT evaluates to "none". If *number_of_characters_to_extract* evaluates to an integer whose value is greater than the size of *text_to_extract_from*, RIGHT evaluates to the entire value of *text_to_extract_from*.

Examples

- RIGHT ("Abcd" , 2)
  Returns cd
- RIGHT ("Abcd" , 6)
  Returns Abcd
- RIGHT ("000000" + NUMBERTOTEXT (TransactionNum), 6)
  If **TransactionNum** contains 123, this example returns 000123. If **TransactionNum** contains 123456789, this example returns 456789.

Related functions

- FIND
- LEFT
- MID

SERIESTOTEXT

You can use the SERIESTOTEXT function to project your input data as a series and to interpret it as a text item for output.

SERIESTOTEXT converts a contiguous or non-contiguous series to a text item.

Syntax:
SERIESTOTEXT (series-object-expression)

Meaning:
SERIESTOTEXT (series_to_convert)

Returns:
A single text item

SERIESTOTEXT returns a text item containing the concatenation of the series of the input argument, including nested delimiters but excluding initiators and terminators.

Examples

In this example, you have the following data that represents bowler information for a bowling league:

Andrews, Jessica:980206:JBC:145:138:177:159

Little, Randy:980116:BBK:175:168

Wayne, Richard:980102:JBC:185:204:179:164:212

Each record consists of the bowler's name, the date of their last game played, a team code and one or more bowling scores. **Record** is defined as a group that is infix-delimited by a colon.

Using the rule:

= SERIESTOTEXT (Score Field:Bowler:Input)

the following results are produced, which is the concatenation of all of the scores for all of the bowlers, even though the scores are not all contiguous within the data:

145138177159175168185204179164212

However, if the rule was changed, for instance, to concatenate the list of scores to the bowler's name:

= BowlerName Field:Bowler:Input + " ->" +
SERIESTOTEXT (Score Field:Bowler:Input)

the following output would be produced:

Andrews, Jessica -> 145138177159

Little, Randy -> 175168

Wayne, Richard -> 185204179164212

In this example, you have an input number that is of variable size, followed by a name. There is no syntax that separates the number from the name. You can define the number as a group with **Byte(s)** as a component and provide a component rule for **Byte(s)**, such as:

ISNUMBER ($)

Based on this, the number can be distinguished from the name. When mapping, collect all the bytes of the number back again. You can use SERIESTOTEXT to do this.

Related functions

- PACKAGE
- TEXT

SQUEEZE

The SQUEEZE function removes consecutive duplicate occurrences of a specified character or characters from a text item.

Syntax:

SQUEEZE (single-text-item , single-text-item)

Meaning:
SQUEEZE (text_to_squeeze , duplicate_characters_to_remove)

Returns:
A single text item

SQUEEZE returns a text item with all the consecutive duplicates of *duplicate_characters_to_remove* removed from *text_to_squeeze*.

Examples

- SQUEEZE (″AB CDE F″, ″ ″)
  Returns: AB CDE F
- SQUEEZE (″Connolly″, ″n″)
  Returns: Conolly

Related functions

- LEAVEALPHA
- LEAVEALPHANUM
- LEAVENUM
- LEAVEPRINT
- SUBSTITUTE

SUBSTITUTE

You can use the SUBSTITUTE function to replace or remove a character. You can also use this function for multiple pairs.

Syntax:
SUBSTITUTE (single-text-expression { , single-text-expression , single-text-expression })

Meaning:
SUBSTITUTE (item_to_convert, one-or-more-text-substitution-pairs)

Where each one-or-more-text substitution-pair is
text_to_change , substitute_text

Returns:
A single text item

SUBSTITUTE returns the text string that results from replacing all instances of the first *text_to_change* with *substitute_text* in *item_to_convert*, then replaces all instances of the second *text_to_change* with the *substitute_text* in the result of the first substitution, and so forth.

Examples

- SUBSTITUTE (″123*456*7″ , ″*″ , ″/″)
  Finds 123*456*7 and returns 123/456/7
- SUBSTITUTE (″120-45-6789″ , ″-″ , ″″)
  Finds 120-45-6789 and returns 120456789
- =SUBSTITUTE (″ABBA″ , ″B″, ″A″, ″A″, ″B″)
  This example illustrates multiple searches for the SUBSTITUTE function.
  The first search-and-replace finds all ″B″s and returns ″A″s: AAAA
  The next search-and-replace finds all ″A″s and returns ″B″s: BBBB

The end result is a return of: BBBB

Related functions

- LEAVEALPHA
- LEAVEALPHANUM
- LEAVENUM
- LEAVEPRINT
- SQUEEZE

TEXT

You can use the TEXT function to convert an object to a text item or when echoing entire data objects to another map.

TEXT converts the content of a group or item object to a text item.

Syntax:

TEXT (single-object-expression)

Meaning:

TEXT (object_to_convert)

Returns:

A single text item

The TEXT function converts *object_to_convert* to a text-item, excluding the initiator and terminator of the input object.

Examples

- TEXT (Record:card)

  Data: #1339X10A,491.38,Green,42x54@

  Returns: 1339X10A,491.38,Green,42x54

  In this example, the group **Record** has an initiator of the pound sign (**#**), a terminator of the at-sign (**@**), and a delimiter of a comma (**,**).

  The initiator and terminator are not included because only the content of the object is converted to text.

Related functions

- DATETOTEXT
- FROMDATETIME
- FROMNUMBER
- NUMBERTOTEXT
- PACKAGE

  Note: TEXT differs from PACKAGE in that it does not include the initiator and terminator of the input object.
- SERIESTOTEXT
- TIMETOTEXT

TEXTTOBCD

The TEXTTOBCD function converts a text item from decimal digits to BCD (Binary Coded Decimal) format.

Syntax:

TEXTTOBCD (single-integer-text-expression)

Meaning:

TEXTTOBCD (text_to_be_converted)

Returns:

A single BCD-formatted text item

TEXTTOBCD converts *text_to_be_converted* (which consists of decimal digits) to BCD format. In this format, each byte contains two decimal digits represented as binary numbers. If there is an odd number of decimal digits in the input, the high-order half-byte of the leftmost output byte will contain the decimal value 15 (hex ″F″).

If anything other than a decimal digit is encountered in the input, TEXTTOBCD returns ″none″.

Examples

- TEXTTOBCD (″1234″)

  Returns the hexadecimal value x'1234'
- TEXTTOBCD (″123A″)

  Returns ″none″
- TEXTTOBCD (″123″)

  Returns the hexadecimal value x'F123'

  In this example, the values shown as input (″123″) are meant to represent character items in the native character set to the machine on which the map is running. On a personal computer, ″123″ would contain the ASCII characters for the digits that have the hexadecimal values ″31″, ″32″, and ″33″. The output, described as ″the hexadecimal value `F123'″, consists of the two binary bytes ″F1″ and ″23″.

  On an IBM mainframe the input string would contain EBCDIC characters for the digits that have the hexadecimal values ″F1″, ″F2″, ″F3″, Âº, but the output would be the same as the personal computer output.

Related functions

- BCDTOHEX
- BCDTOINT
- BCDTOTEXT

TEXTTONUMBER

Use TEXTTONUMBER to convert text to a number.

The TONUMBER function provides greater flexibility for specifying the format of the text item that is to be converted to a number.

Syntax:

TEXTTONUMBER (single-text-expression)

Meaning:
TEXTTONUMBER (text_to_convert_to_number)

Returns:
A single character number

The *text_to_convert_to_number* must be in integer or ANSI-formatted (floating point) presentation. The resulting number looks like the input argument, however, nonsignificant zeroes to the right of the decimal separator will be truncated. If the input argument is in error (for example, it is not a recognizable as a valid number), the result is ″none″.

When specified as in ANSI-formatted presentation, the text string must meet the following requirements:

- The decimal point can be a period, a comma, or ″none″.
- The leading sign can be a plus sign, a minus sign, or ″none″.
- No thousands separator is allowed.

Examples

- TEXTTONUMBER (OrderQty)
  Returns **OrderQty** as a character number item

Related functions

- DATETOTEXT
- FROMNUMBER
- NUMBERTOTEXT
- TEXTTODATE
- TEXTTOTIME
- TIMETOTEXT

TEXTTOTIME

Use TEXTTOTIME when you want to convert an object defined as text that is in HHMM or HHMMSS presentation, to an item defined as time. For greater flexibility, use the TODATETIME function for specifying the format of the text item that is to be converted to a date/time.

Syntax:
TEXTTOTIME (single-text-expression)

Meaning:
TEXTTOTIME (text_to_convert_to_time)

Returns:
A single time

The *text_to_convert_to_time* must be in HHMM or HHMMSS presentation. HH is a two-digit hour in a 24-hour format. If the result is being assigned to a time object, the resulting time looks like the output object. Otherwise, the resulting time looks like the input argument. If the input argument is in error (for example, it is not a valid time), the result is ″none″.

Examples

- TEXTTOTIME (CallTime)

  Returns **CallTime** as a time item

Related functions

• DATETOTEXT	• TEXTTONUMBER
• FROMDATETIME	• TIMETOTEXT
• NUMBERTOTEXT	• TODATETIME
• TEXTTODATE	

TIMETOTEXT

You can use the TIMETOTEXT function to perform text concatenation. For greater flexibility, use the FROMDATETIME function for specifying the format of the resulting text item.

TIMETOTEX converts a time object or expression to a text item.

Syntax:

TIMETOTEXT (single-time-expression)

Meaning:

TIMETOTEXT (time_to_convert_to_text)

Returns:

A single text item

If *time_to_convert_to_text* is a time object name, this returns the time as a text item formatted according to the presentation of the input date object.

If *time_to_convert_to_text* is a time expression produced by a function, this returns the time as a text item formatted according to the presentation of the output argument of that function.

Examples

- TIMETOTEXT (LeadTime)

  In this example, **LeadTime** is converted from a time to text. If **LeadTime** has an HH:MM presentation, the resulting text item will be of that presentation.

- TIMETOTEXT (CURRENTDATETIME (″{HH:MM:SS}″))

  Here, CURRENTDATETIME evaluates and returns a time in HH:MM:SS format. Then, TIMETOTEXT evaluates and returns a text string that is that time in HH:MM:SS format.

Related functions

- DATETOTEXT
- FROMDATETIME
- NUMBERTOTEXT
- TEXTTODATE
- TEXTTONUMBER
- TEXTTOTIME

- TODATETIME

TODATETIME

The TODATETIME function converts a text string of a specified format to a date-time item.

Syntax:

```
TODATETIME ( single-character-text-expression
 [ , single-text-expression ] )
```

Meaning:

TODATETIME (text_to_convert [, date_time_format_string])

Returns:

A single character date item

TODATETIME returns the date-time that corresponds to the value specified by *text_to_convert*, which is in the format specified by *date_time_format_string*. If *date_time_format_string* is not specified, it will be assumed that *text_to_convert is* in [CCYYMMDDHH24MMSS] format.

The *date_time_format_string* must conform to the date-time format strings as described in ″Format strings″.

Examples

- TODATETIME (″05/14/1999@10:14pm″ , ″{MM/DD/CCYY}@{HH12:MMAM/PM}″)

  In this example, a text string containing a date and time is converted to a date-time item.

- RptDate = TODATETIME (RIGHT (GETRESOURCENAME(), 8) , ″CCYYMMDD″)

  Assume that you receive a file that contains historical data. The name of the file identifies the date of the historical data. For example, a filename of **19960424** indicates that the data was produced on April 24, 1996. To map this date to **RptDate**, the TODATETIME function could be used with the RIGHT and GETRESOURCENAME functions.

Related functions

- CURRENTDATE
- CURRENTDATETIME
- CURRENTTIME
- TEXTTODATE
- TEXTTOTIME

TONUMBER

The TONUMBER function converts a text string of a specified format to a number.

Syntax:

```
TONUMBER ( single-character-text-expression [ , single-text-expression ] )
```

Meaning:

TONUMBER (text_to_convert [, number_format_string])

Returns:

A single character number item

TONUMBER returns the number that corresponds to the value specified by *text_to_convert*, which is in the format specified by *number_format_string*. If *number_format_string* is not specified, it will be assumed that *text_to_convert* is in ANSI decimal format (for example, "{L-####['.'##]}").

The *number_format_string* must conform to the number format strings as described in the "Format strings" section.

Examples

- TONUMBER(*text_to_convert*, "{L+'$'#','###}")

  L+'$' indicates the leading dollar sign is positive. That leading sign and the comma separators are removed when the text is converted to a number.

  Input String: $123,000,000

  Output: 123000000

- TONUMBER(*text_to_convert*, "{####T-}")

  Four number signs are required for each whole number, regardless of the actual number of digits in the number.

Input string:	Output:	Note:
12345-	-12345	The output becomes a negative number.
67890	67890	No change occurs.
345-	-345	The output becomes a negative number.

- TONUMBER(*text_to_convert*, "{####T+'K'-}")

  If an invalid character, such as an X, is encountered, nothing is returned.

  If a **K** is encountered, it is treated as a positive indicator.

Input string:	Output:	Note:
11212-	-11212	The output becomes a negative number.
67890X		The X is an invalid character. No number is returned.
54354	54354	No change occurs.
34567K	34567	The K is recognized as a positive sign. The character is removed and the number is returned as a positive.
345-	-345	The output becomes a negative number.

- TONUMBER(*text_to_convert*, "{L-'('#','###T-')'}")

  The parentheses indicating a negative number are removed and replaced with a negative sign.

  Comma separators are removed when the text is converted to a number.

Input string:	Output:	Note:
(12,345)	-12345	The output becomes a negative number. The comma separator is removed.

Input string:	Output:	Note:
67,890	67890	The comma separator is removed.
(345)	-345	The output becomes a negative number.

- TONUMBER(*text_to_convert*, "{#[',']###['.'##5]T+'K'-}")

  The optional comma separators are removed, but the decimal points and decimal values are retained.

Input string:	Output:	Note:
54,345.098	54354.098	The comma separator is removed.
67890.0X		The X is an invalid character. No number is returned.
11213-	-11213	The output becomes a negative number.
34567K	34567	The K is recognized as a positive sign. The character is removed and the number is returned as a positive.
345.1-	-345.1	The output becomes a negative number.

Related functions

- DATETONUMBER
- FROMNUMBER
- NUMBERTODATE
- NUMBERTOTEXT

TRIMLEFT

You can use the TRIMLEFT function to remove spaces or a text string at the beginning of some text.

TRIMLEFT removes leading characters from a text item.

Syntax:

TRIMLEFT (single-text-expression [, single-text-expression])

Meaning:

TRIMLEFT (item_to_trim [, text_to_trim])

Returns:

A single text item

TRIMLEFT removes leading characters that match *text_to_trim* from *item_to_trim* and returns the result as a single text item. If *text_to_trim* is not specified, leading spaces are removed from *item_to_trim*.

Examples

- TRIMLEFT (" abc")

  Returns: abc
- TRIMLEFT ("000345" , "0")

  Returns: 345
- TRIMLEFT (LastName Column:Row , "<WSP>")

Returns the content of **LastName Column** after removing all leading whitespace characters (space, horizontal tab, carriage return, or line feed characters).

Related functions

• FILLLEFTP	• LEAVEPRINT
• FILLRIGHT	• SQUEEZE
• LEAVEALPHA	• SUBSTITUTE
• LEAVEALPHANUM	• TRIMRIGHT
• LEAVENUM	

TRIMRIGHT

You can use the TRIMRIGHT function to remove spaces or a text string at the end of text.

TRIMRIGHT removes trailing characters from a text item.

Syntax:

TRIMRIGHT (single-text-expression [, single-text-expression])

Meaning:

TRIMRIGHT (item_to_trim [, text_to_trim])

Returns:

A single text item

TRIMRIGHT removes trailing characters that match *text_to_trim* from *item_to_trim* and returns the result as a single text item. If *text_to_trim* is not specified, trailing spaces are removed from *item_to_trim*.

Example

- TRIMRIGHT (″abc ″)

  Returns: abc
- TRIMRIGHT (″Cat in the Hat!?!?!?″ , ″!?″)

  Returns: Cat in the Hat
- TRIMRIGHT (LastName Column:Row)

  Returns the content of **LastName Column** after removing all trailing spaces.

Related functions

• FILLLEFTP	• LEAVEPRINT
• FILLRIGHT	• SQUEEZE
• LEAVEALPHA	• SUBSTITUTE
• LEAVEALPHANUM	• TRIMLEFT
• LEAVENUM	

UPPERCASE

The UPPERCASE function converts text to all uppercase characters.

Syntax:

UPPERCASE (single-text-expression)

Meaning:

UPPERCASE (text_to_convert)

Returns:

A single text item

UPPERCASE produces a text item in which each byte in *text_to_convert* has been converted to uppercase. Any numeric or symbolic characters in *text_to_convert* remain unchanged.

Examples

- UPPERCASE (″abC123!″)

  Returns ABC123!

Related functions

- ISLOWER
- ISUPPER
- LOWERCASE

WORD

You can use the WORD function to parse a text item that is delimited by some character, such as a space or a comma.

WORD returns the characters between two user-defined separators within a text item. The separators are counted from left to right when the third argument is positive, and from right to left when the third argument is negative, enabling the function to search from either end of the item.

Syntax:

WORD (single-text-expression , single-text-expression , single-integer-expression)

Meaning:

WORD (text_to_search , word_separator, number_of_word_to_get)

Returns:

A single text item

WORD returns the characters (word) between the *n*th-1 and *n*th *word_separator*, where *n* corresponds to *number_of_word_to_get*.

Define the separator (*word_separator*) and specify the number of the occurrence of that separator. The WORD function returns the characters between the *nth-1* and *n*th separators in the delimited text item.

The separator is case-sensitive.

Examples

The following examples assume that a file exists named **Letter** with two text objects named **Line1** and **Line2** as follows:

```
Line1:Congratulations, Mr Brown! You're a winner!;

Line2:You may have already won 1 million dollars!;
```

- WORD (Line1:Letter , " " , 3)

  Returns: `Brown!`

  The exclamation point is returned because it is read as a character within the word before the separator (a space).
- WORD (Line1:Letter , " " , 6)

  Returns: `winner!`

  If the *n*th separator is missing and the *nth-1* separator exists, the function returns the characters between the last separator and the end of the delimited text item. The separator is a space; "`winner!`" (including the exclamation point) is the sixth word.
- WORD (Line2:Letter , "!" , 3)

  Returns "none"

  Both *nth* and *nth-1* separators are missing. In this example, there is only one separator, located at the end of the text object. As a result, there is no third word because the function sees everything before "!" as the first word.
- WORD (Line1:Letter , " " , 1)

  Returns: `Congratulations,`

  If *n-1* = 0, the function returns the characters between the beginning of the delimited text item and the first separator.
- WORD (Line1:Letter , " " , -1)

  Returns: `winner!`

  If *n+1* = 0, then the function returns the characters between the end of the text item and the last separator.

Related functions

- FIND
- MID

Chapter 17. XML functions

VALIDATE

The VALIDATE function validates the input XML.

This function validates XML input, given as a text stream or uniform resource locator (URL), against the provided XML Schema, returning 0 if the validation succeeded or -1 otherwise.

Syntax:
VALIDATE (single-text-expression, single-text-expression)

Meaning:
VALIDATE (xml_url_or_xml_fragment, xml_schema_url)

Returns:
A single number

Examples

- `NUMBERTOTEXT( xmllib->VALIDATE( "ipo.in.xml", "http://www.example.com/IPO ipo.xsd" ) )`

  Returns `0` when validation of the ipo.in.xml file succeeds and returns `-1` otherwise.
- `VALID( NUMBERTOTEXT( xmllib->VALIDATE( PACKAGE(source), "ipo.xsd" ) ), LASTERRORMSG() )`

  Returns `0` when validation of the input XML fragment succeeds or returns a validation error message otherwise.

XPATH

The XPATH function queries the input XML.

This function queries the XML input, given as a text stream or URL, using the given XPath expression and context, returning the result of the evaluation.

Syntax:
XPATH (single-text-expression, single-text-expression, single-text-expression)

Meaning:
XPATH (xml_url_or_xml_fragment, xpath_expression, context_expression)

Returns:
A single text item

Examples

- `xmllib->XPATH( "ipo.in.xml", "./order//item[1]/shipDate", "/ipo:purchaseOrders " )`

  Returns the content of the shipDate element from the ipo.in.xml file.
- `xmllib->XPATH( PACKAGE(source), "/ipo:purchaseOrders/order/items/item[1]/ shipDate", "/" )`

  Returns the content of the shipDate element from the input XML fragment.

XSLT

The XSLT function applies an XSLT transformation.

This function applies an XSLT transformation, expressed as a text stream or URL, to the given XML input, returning the result of the transformation.

Syntax:

XSLT (single-text-expression, single-text-expression)

Meaning:

XSLT (xml_url_or_xml_fragment, xslt_url_or_xslt_fragment)

Returns:

A single text item

Examples

- `xmllib->XSLT( "ipo.in.xml", "ipo.xsl" )`

  Applies the XSLT transformation defined in the ipo.xsl file to the ipo.in.xml file.
- `xmllib->XSLT( PACKAGE( source ), "ipo.xsl" )`

  Applies the XSLT transformation defined in the ipo.xsl file to the input XML fragment.

Chapter 18. Custom functions

From the Type Designer or Map Designer, you can create custom functions to call external libraries.

While the EXIT function is often used to call external libraries, the EXIT function is limited to support only text objects and is operating system dependent. When you create a custom function to use in a map rule, the function is operating system independent and supports text, number, and date-time objects.

Similar to the regular functions that are shipped with the product, a maximum of four parameters are supported per argument.

Creating a custom function

To create your own functions, a C or C++ development tool is required, or the Java Native Interface (JNI) if you are working in a Java environment.

1. From a component rule in the Type Designer or from a map rule in the Map Designer, right-click and select the **Insert Function** option. (Additionally in the Map Designer, if you are not in a map rule, you can create a new function by selecting **Rules** → **New Function**.)
2. From the Insert Function window, select **Design**. The Custom Function Modules window is displayed.
3. In the **Path** field, enter the path of where to create the external library. This should always be in the *install_dir*/function_libs directory.
4. In the **Name** field, enter a name for the library.
5. Click **Add** to add a function to the library. The Function Specifics window is displayed.
6. In the **Name** field, enter a name for the function.
7. For **Return Type**, select the function return type from the drop-down list. You can select up to four function parameters from the respective drop-down list. Choices include Boolean, date, number, text, time, and byte stream.
8. In the space provided, enter a description for your function. This information will be displayed in the Insert Function window of both Designers.
9. Click **OK** to validate the selected parameters and close the Function Specifics window.
10. Click **Generate**. The Designer generates a collection of operating system specific makefiles and definition files that provide the framework for the function you are creating.
11. Go to the *install_dir*/function_libs directory to view the results. As a result of the generation process, framework files are created for each platform that WebSphere Transformation Extender supports. You can find the following platform specific makefiles and definition files in the *install_dir*/function_libs/*your_new_lib* directory:
    - makefile (Windows)
    - aix.mak
    - hp.mak
    - hpi32.mak
    - linux.mak

- os390.mak
- sun.mak
- *your_new_lib*.def (definition file)
- *your_new_lib*.c (C file with the functions defined)

12. Now you must modify the framework that was generated by the Designer. The following functions with the parameter information that you selected were exported to the .c file:
    - GetFunctionCount
    - GetFunctionName
    - GetInputParameter
    - GetReturnType
    - GetParameterCount
    - GetFunctionDesc

    To complete the new function, open the .c file and add your programming code for the applicable function(s) provided.

    Use the .c file to build your dynamic link library (DLL) and then place the DLL in the *install_dir*/function_libs/*your_new_lib* directory. (All custom designed libraries and functions must be placed in the *install_dir*/function_libs directory.)

    The new library name is listed under **Category** in the Insert Function dialog, and the new function that you created is placed in the list of functions and will remain available for future use from both the Type Designer and Map Designer applications.

The following example displays how to implement a custom function called SIN.

```
void ConvertToBytes(double  returnValue, LPEXITPARAM lpep)
{
  double value = 0.0;
  int decimal = 2,   sign = 0, j = 0, k = 0;
  char byString[100];
  char* lpbyData = _fcvt(returnValue, 7, &decimal, &sign );
  memset(byString, 0, sizeof(byString));

  if (sign)
       byString[j++] = '-';

  if (decimal  <= 0)
  {
   byString[j++] = '0';
   byString[j++] = '.';
    while (decimal != 0)
    {
     byString[j++] = '0'; decimal++;
    }
  }
  else if (decimal > 0)
  {
    while (decimal != 0)
    {
     byString[j++] = lpbyData[k++]; decimal--;
    }
     byString[j++] = '.';
  }
  while (lpbyData[k])  byString[j++] = lpbyData[k++];

  byString[j] = '\0';

  if (NULL == (lpep->lpDataFromApp = GlobalAllocPtr
```

```
     (GHND, j + 1)))
  {
  lpep->nReturn = -1;
  lstrcpy(lpep->szErrMsg, "Memory allocation failed in Alternate");
  return;
  }

  memcpy(lpep->lpDataFromApp, byString, j);
  lpep->dwFromLen =  j;
  lpep->lpDataFromApp[j++] = '\0';
}


void CALLBACK EXPORT SIN(LPEXITPARAM lpep)
{
  double value = 0.0;
  double returnValue = 0.0;
  LPEXITPARAMEXTENDED lpExtended = NULL;

  if (lpep->dwSize != sizeof(EXITPARAM))
  {
   return;
  }

  lpExtended = (LPEXITPARAMEXTENDED)lpep->lpv;
value = atof(lpExtended->lpFirstInputParameter);

returnValue = sin(value);
ConvertToBytes(returnValue, lpep);
  lpep->wCleanupAction = GetReturnType("SIN");
lstrcpy(lpep->szErrMsg, "SIN function was successful");

return;
}
```

At runtime, all custom designed libraries and functions that your maps use must be in the *install_dir*/function_libs directory. When you deploy a map that uses a custom function to a remote host, the library is not transferred. Therefore you must manually copy the custom function library to the install_dir/function_libs directory on the remote host.

Chapter 19. Date and time format strings

You can use the listed format strings for numbers, dates and times in functions such as the CURRENTDATETIME, FROMNUMBER, TONUMBER, FROMDATETIME, and TODATETIME.

Create custom date and time formats by using the symbols based on the given format strings.

Time units

Symbol
Description

HH24 Hour (based on a 24-hour clock) in two digits (00 to 23)

H24 Hour (based on a 24-hour clock) in one or two digits as needed (0 to 23)

HH12 Hour (based on a 12-hour clock) in two digits (1-12)

H12 Hour (based on a 12-hour clock) in one or two digits as needed (1 to 12)

MM Minute in two digits (00 to 59)

M Minute in one or two digits as needed (0 to 59)

SS Seconds in two digits (00 to 59)

S Second in one or two digits as needed (0 to 59)

AM/PM
Meridian (AM/PM)

ZZZ Three character abbreviation for time zone (EST, and so on)

+/-ZZZZ
Hours and minutes before or after Greenwich Mean Time (GMT), also known as Coordinated Universal Time (UTC)

+/-ZZ:ZZ
4-digit time where the format is a 2-digit hour and 2-digit minute, separated by a colon.

+/-ZZ[:ZZ]
4-digit time where the format is a 2-digit hour and an optional 2-digit minute, separated by colon.

+/-ZZ[ZZ]
4-digit time where the format is a 2-digit hour and an optional 2-digit minute.

Date units

Symbol
Description

CCYY Full year including century (2001)

YY Last two digits of the year (00-99)

MM Month of the year in two numeric digits (01-12)

M Month of the year in one or two numeric digits as needed (1 to 12)

MON First three letters of the month (Jan to Dec)

MONTH
Full name of the month (January to December)

DDD Day of the year in three numeric digits (001 to 366)

DD Day of the month in two numeric digits (01 to 31)

D Day of the month in one or two numeric digits as needed (1 to 31)

DY First three letters of the weekday (Sun to Sat)

DAY Full name of the weekday (Sunday to Saturday)

WW Week of year (1-52)

Qn Quarter of the year (Q1-Q4)

EEYY Emperor's year, long form

EY Emperor's year, short form

Binary date and time format strings

Sub-String Name
Sub-String Value

Binary DateTime
["{" + Date + "}"] + ["{" + Time + "}"]

Date CCYYMMDD

YYMMDD

CCYYDDD

YYDDD

Time HH24MMSS

HH24MM

Japanese date and time format strings

Sub-String Name
Sub-String Value

Japanese DateTime
"{" + DateTime + "}"

"{" + DateTime + "}" + Separator(1) + "{" + DateTime + "}"

"{" + DateTime + "}" + "[" + Separator(1) + "{" + DateTime + "}" + "]"

DateTime
Date

Time

Separator
Separator is optional. If specified, it can be up to 120 bytes, composed of non-alphabetic characters. Symbol table values, such as <CR>, can be used to indicate non-printable characters.

The Separator is required if the second DateTime option is used and the format string of the first DateTime ends in a variable sized specification - EY, M, or D for Date or H24, H12, M, or S for Time.

Japanese date format strings

Sub-String Name
Sub-String Value

Japanese Date
Year + MonthSet(1)

Year CCYY

YY

EEYY

EY

MonthSet
Separator(1) + ″MM″ + DayOfMonth(1)

″[″ + Separator(1) + ″MM″ + DayOfMonth(1) +]

Separator(1) + M + DayOfMonth(1)

[+ Separator(1) + M + DayOfMonth(1) +]

DayOfMonth
Separator(1) + ″DD″ + WeekDay(1)

″[″ + Separator(1) + ″DD″ + WeekDay(1) + ″]″

Separator(1) + ″D″ + WeekDay(1)

″[″ + Separator(1) + ″D″ + WeekDay(1) + ″]″

WeekDay
Separator(1) + ″DY″

″[″ + Separator(1) + ″DY″ + ″]″

Separator(1) + ″DAY″

″[″ + Separator(1) + ″DAY″ + ″]″

Separator
Separator is optional. If specified, it can be up to 120 bytes composed of non-alphabetic characters. Symbol table values, such as <CR>, can be used to indicate non-printable characters.

MonthSet Separator required if Year is EY

DayOfMonth Separator required if MonthSet is M

WeekDay Separator required if DayOfMonth is D

Japanese time format strings

Sub-String Name
Sub-String Value

Japanese Time
Meridian(1) + HMS

Meridian
″AM/PM″

HMS ″HH24″ + MinuteSet(1)

″H24″ + MinuteSet(1)

″HH12″ + MinuteSet(1)

″H12″ + MinuteSet(1)

MinuteSet

Separator(1) + ″MM″ + SecondSet(1)

″[″ + Separator(1) + ″MM″ + SecondSet(1) + ″]″

Separator(1) + ″M″ + SecondSet(1)

″[″ + Separator(1) + ″M″ + SecondSet(1) + ″]″

SecondSet

Separator(1) + ″SS″

″[″ + Separator(1) + ″SS″ + ″]″

Separator(1) + ″S″

″[″ + Separator(1) + ″S″ + ″]″

Separator

Separator is optional. If specified, it can be up to 120 bytes, composed of non-alphabetic characters. Symbol table values, such as <CR> can be used to indicate non-printable characters.

MinuteSet Separator required if HMS is H24 or H12

SecondSet Separator required if MinuteSet is M

Western date and time format strings

Substring Name
Substring Value

Western DateTime

″{″ +DateTime + ″}″

″{″ +DateTime + ″}″ + Separator(1) + ″{″ + DateTime + ″}″

″{″ +DateTime + ″}″ + ″[″ + Separator(1) + ″{″ + DateTime + ″}″ + ″]″

DateTime

Date

Time

Separator

Separator is optional. If specified, it can be up to 120 bytes, composed of non-alphabetic characters. Symbol table values, such as <CR>, can be used to indicate non-printable characters.

The Separator is required if the second DateTime option is used and the format string of the first DateTime ends in a variable sized specification - M, MONTH, D, or DAY for Date or H24, H12, M, or S for Time.

Western date format strings

Sub-String Name
Sub-String Value

Date DateUnit + DatePart2(1)

DatePart2

Separator(1) + DateUnit + DatePart3(1)

″[″ + Separator(1) + DateUnit + DatePart3(1) + ″]″

DatePart3

Separator(1) + DateUnit + DatePart4(1)

″[″ + Separator(1) + DateUnit + DatePart4(1) + ″]″

DatePart4

Separator(1) + DateUnit

″[″ + Separator(1) + DateUnit + ″]″

Separator

Separator is optional. If specified, it can be up to 120 bytes, composed of non-alphabetic characters. Symbol table values, such as <CR>, can be used to indicate non-printable characters.

When DateUnit has a variable length (M, MONTH, D, DAY), a Separator must follow that DateUnit if data follows.

Western time format strings

Sub-String Name

Sub-String Value

Western Time

Hours + MinutesSet(1) + Meridian(1) + Zone(1)

Hours HH24

H24

HH12

H12

MinutesSet

Separator(1) + ″MM″ + SecondSet(1)

″[″ + Separator(1) + ″MM″ + SecondSet(1) + ″]″

Separator(1) + ″M″ + SecondSet(1)

″[″ + Separator(1) + ″M″ + SecondSet(1) + ″]″

SecondSet

Separator(1) + ″SS″ + FractionSet(1)

″[″ + Separator(1) + ″SS″ + FractionSet(1) + ″]″

Separator(1) + ″S″ + FractionSet(1)

″[″ + Separator(1) + ″S″ + FractionSet(1) + ″]″

FractionSet

Separator(1) + MinPlaces + ″-″ + MaxPlaces

″[″ + Separator(1) + MinPlaces + ″-″ + MaxPlaces + ″]″

MinPlaces

An integer from 0 to 9

MaxPlaces

An integer from 0 to 9 (must be less than MinPlaces)

Meridian

″AM/PM″

″[AM/PM]″

Zone ″+/-ZZZZ″

″ZZZ″

″[+/-ZZZZ]″

″[ZZZ]″

Separator

Separator is optional. If specified, it can be up to 120 bytes, composed of non-alphabetic characters. Symbol table values, such as <CR>, can be used to indicate non-printable characters.

MinuteSet Separator required if Hours is H24 or H12

SecondSet Separator required if MinuteSet is M

FractionSet Separator required if SecondSet is S

Chapter 20. Number format strings

You can create custom number formats by using the given format strings.

Decimal

″{″ + Leading-Sign(1) + Whole# + Fraction + Trailing-Sign(1) + ″}″

Integer

″{″ + Leading-Sign(1) + Whole# + Trailing-Sign(1) + ″}″

Leading sign format strings

Sub-String Name	Sub-String Value	Positive	Negative
Leading-Sign	″L″ + Positive + Negative + Zero(1) -or-	Req	Req
	″L″ + ″[″ + Positive + ″]″ + Negative + Zero(1) -or-	Opt	Req
	″L″ + Positive + ″[″ + Negative + ″]″ + Zero(1) -or-	Req	Opt
	″L″ + Positive + Zero(1) -or-	Req	-
	″L″ + Negative + Zero(1)	-	Req

Trailing sign format strings

Sub-String Name	Sub-String Value	Positive	Negative
Trailing-Sign	″T″ + Positive + Negative + Zero(1)	Req	Req
	″T″ + ″[″ + Positive + ″]″ + Negative + Zero(1)	Opt	Req
	″T″ + Positive + ″[″ + Negative + ″]″ + Zero(1)	Req	Opt
	″T″ + Positive + Zero(1)	Req	-
	″T″ + Negative + Zero(1)	-	Req

Substring format strings

Substring Name	Substring Value	Meaning
Positive	″+″ + Value(1)	
Negative	″-″ + Value(1)	

Substring Name	Substring Value	Meaning
Value	A text-string enclosed in single-quotes ’. Value has a release character of / if the text contains any single quote ’ or forward slash / characters.	The sign value of the previous sign-indicator. If Value is not used, the default is + for positive numbers and - for negative numbers.
Zero	″Z″ + Value ″[″ + ″Z″ + Value + ″]″	Specifies the required leading sign value if the number is zero. If Zero is not used, there is no sign associated with a zero. Specifies the optional leading sign value if the number is zero. If Zero is not used, there is no sign associated with a zero.

Whole number and fraction format strings

Substring Name	Substring Value	Meaning
Whole#	MinDigits(1) + ″#″ + Value(1) + ″###″ + MaxDigits(1) -or- MinDigits(1) + ″#″ + ″[″ + Value(1) + ″]″ + ″###″ + MaxDigits(1)	Specifies the thousands separator and the range of whole number digits. If min-digits is not used, the default is zero. If max-digits is not used, the default is ″S″. If a ThousandsItem is specified, a Value must be present as the default for the syntax item.
Fraction	″V″ + ImpliedPlaces - or - Value + MinDigits(1) + ″##″ + MaxDigits(1) - or - ″[″ + Value + MinDigits(1) + ″##″ + MaxDigits(1) + ″]″	Specifies the decimal to be implicit and implied places specifies where the intended decimal separator is to be placed. Specifies the value of the decimal separator to be required and the range of fraction digits. If min-digits is not used, default is zero. If max-digits is not used, default is ″S″ Specifies the value of the decimal separator to be optional if there is no fractional portion of the number. It also specifies the range of fraction digits. If min-digits is not used, default is zero. If max-digits is not used, default is ″S″

Chapter 21. RUN function return codes

The RUN function return codes and messages may result when using the RUN function. Return codes and messages are returned when the particular activity completes. Return codes and messages may also be recorded as specified in the audit logs, trace files, execution summary files, etc.

The following table lists the return codes and messages that can result when using the RUN function.

Return Code
Message

50 *Memory allocation failure*

Occurs when memory fails.

51 *Card override failure*

Occurs when memory fails.

52 I/O *initialization* failure

Occurs when memory fails.

53 *Open audit failure*

The audit log file is not accessible.

54 *No command line*

There is nothing to process.

55 *Recursive command files*

More than one command file is included in the command line.

56 *Invalid command line option -x*

The option is invalid for the command.

57 *Invalid `W' command line option*

The Work file option is invalid.

58 *Invalid `B' command line option*

The Batch (close) file option is invalid.

59 *Invalid `R' command line option*

The Refresh Rate option is invalid.

60 *Invalid `A' command line option*

The Audit option is invalid.

61 *Invalid `P' command line option*

The Paging option is invalid

62 *Invalid `Y' command line option*

The General I/O Retry option is invalid.

63 *Invalid `T' command line option*

The Trace option is invalid.

64 *Invalid `G' command line option*

The Ignore option is invalid

65 *Invalid `I' command line option for input x*

The Source option is invalid for the identified input.

66 *Invalid size in echo command line for input x*

The size specified using the Size option is greater than memory allowed.

67 *Invalid adapter type in command line for input x*

The adapter is not of a known adapter type. Includes -IM*xxx* where *xxx* is an unknown adapter alias.

68 *Invalid `O' command line option for output x*

The target option is invalid for output x. The number of characters between the single quotes that represent the options for an adapter exceed 258 characters in the adapter override.

69

```
Invalid adapter type in command line for output x
```

The adapter is not of a known adapter type. Includes -OMxxx where xxx is an unknown adapter alias.

70 *Command line memory failure*

Occurs when memory is exceeded during echo or override card commands.

71 *Invalid `D' command line option*

The Date option is invalid.

72 *Invalid `F' command line option*

The Failure option is invalid.

73 *Resource manager failure*

(Launcher only) The resource manager is not used, possibly a memory failure.

74 *Invalid `Z' command line option*

The Ignore option is invalid.

75 *Adapter failed to get data on input*

Enable the adapter trace to record the adapter activity to discover the cause of the error.

76 *Adapter failed to put data on output*

Enable the adapter trace to record the adapter activity to discover the cause of the error.

77 *Invalid map name*

This message can occur in two different cases. First, this message occurs when the map name specified on the command line is more than 32 characters long. Also, this message can occur when there is an error in the command line such that text for another execution command is

erroneously being interpreted as the map name. For example, in the command line below, the number representing the size of the echoed data is missing.

```
mymap.mmc -IE1S HereIsMyDataButIForgotToSpecifyTheSize -AED
```

Because the size is missing, it is interpreted to be 0, such that there is no echoed data. The next string encountered on the command line

```
(HereIsMyData...)
```

Because it does not start with a hyphen (-), it is assumed to be the name of the next map to execute. Because the text is longer than 32 characters, the *Invalid Map Name* message is returned.

Chapter 22. Character set codes for CPACKAGE, CSERIESTOTEXT, and CTEXT

The second argument of the CPACKAGE, CSERIESTOTEXT, and CTEXT functions specifies the character set of the output of the function. The value of the second argument (the character set of the object content), must be a valid character set code.

All actions performed that use the text string resulting from one of these functions (CPACKAGE, CSERIESTOTEXT, or CTEXT) treat the text string as being of the specified character set–it is not automatically treated as Native.

Character set code	Data language
ASCII	ASCII
BOCU-1	BOCU-1
CESU-8	CESU-8
CIIKANJI	CII Kanji
EBCDIC	EBCDIC
ebcdic-xml-us	ebcdic-xml-us
EUC	EUC
gb18030	gb18030
HZ	HZ-GB-2312
ibm-1006_P100-1995	ibm-1006
ibm-1025_P100-1995	ibm-1025
ibm-1026_P100-1995	ibm-1026
ibm-1047_P100-1995	ibm-1047
ibm-1047_P100-1995,swaplfnl	ibm-1047-s390
ibm-1051_P100-1995	hp-roman8
ibm-1089_P100-1995	Arabic
ibm-1097_P100-1995	ibm-1097
ibm-1098_P100-1995	ibm-1098
ibm-1112_P100-1995	ibm-1112
ibm-1122_P100-1999	ibm-1122
ibm-1123_P100-1995	ibm-1123
ibm-1124_P100-1996	ibm-1124
ibm-1125_P100-1997	ibm-1125
ibm-1129_P100-1997	ibm-1129
ibm-1130_P100-1997	ibm-1130
ibm-1131_P100-1997	ibm-1131
ibm-1132_P100-1998	ibm-1132
ibm-1133_P100-1997	ibm-1133
ibm-1137_P100-1999	ibm-1137
ibm-1140_P100-1997	ibm-1140 (ebcdic-us-37+euro)

Character set code	Data language
ibm-1140_P100-1997,swaplfnl	ibm-1140-s390
ibm-1141_P100-1997	ibm-1141 (ebcdic-de-273+euro)
ibm-1142_P100-1997	ibm-1142 (ebcdic-dk/no-277+euro)
ibm-1142_P100-1997,swaplfnl	ibm-1142-s390
ibm-1143_P100-1997	ibm-1143 (ebcdic-fi/se-278+euro)
ibm-1143_P100-1997,swaplfnl	ibm-1143-s390
ibm-1144_P100-1997	ibm-1144 (ebcdic-it-280+euro)
ibm-1144_P100-1997,swaplfnl	ibm-1144-s390
ibm-1145_P100-1997	ibm-1145 (ebcdic-es-284+euro)
ibm-1145_P100-1997,swaplfnl	ibm-1145-s390
ibm-1146_P100-1997	ibm-1146 (ebcdic-gb-285+euro)
ibm-1146_P100-1997,swaplfnl	ibm-1146-s390
ibm-1147_P100-1997	ibm-1147 (ebcdic-fr-297+euro)
ibm-1147_P100-1997,swaplfnl	ibm-1147-s390
ibm-1148_P100-1997	ibm-1148 (ebcdic-international+euro)
ibm-1148_P100-1997,swaplfnl	ibm-1148-s390
ibm-1149_P100-1997	ibm-1149 (ebcdic-is-871+euro)
ibm-1149_P100-1997,swaplfnl	ibm-1149-s390
ibm-1153_P100-1999	ibm-1153
ibm-1153_P100-1999,swaplfnl	ibm-1153-s390
ibm-1154_P100-1999	ibm-1154
ibm-1155_P100-1999	ibm-1155
ibm-1156_P100-1999	ibm-1156
ibm-1157_P100-1999	ibm-1157
ibm-1158_P100-1999	ibm-1158
ibm-1160_P100-1999	ibm-1160
ibm-1162_P100-1999	ibm-1162
ibm-1164_P100-1999	ibm-1164
ibm-1168_P100-2002	KOI8-U
ibm-1250_P100-1995	ibm-1250
ibm-1251_P100-1995	ibm-1251
ibm-1252_P100-2000	ibm-1252
ibm-1253_P100-1995	ibm-1253
ibm-1254_P100-1995	ibm-1254
ibm-1255_P100-1995	ibm-1255
ibm-1256_P110-1997	ibm-1256
ibm-1257_P100-1995	ibm-1257
ibm-1258_P100-1997	ibm-1258
ibm-12712_P100-1998	ebcdic-he
ibm-12712_P100-1998,swaplfnl	ibm-12712-s390
ibm-1276_P100-1995	ibm-1276 (Adobe Standard Encoding)

Character set code	Data language
ibm-1363_P110-1997	ibm-1363_P110-1997
ibm-1363_P11B-1998	ibm-1363 (korean)
ibm-1364_P110-1997	ibm-1364
ibm-1371_P100-1999	ibm-1371
ibm-1373_P100-2002	ibm-1373_P100-2002
ibm-1375_P100-2003	Big5-HKSCS (IBM)
ibm-1383_P110-1999	EUC-CN
ibm-1386_P100-2002	ibm-1386-P100-2002
ibm-1388_P103-2001	ibm-1388
ibm-1390_P110-2003	ibm-1390
ibm-1399_P110-2003	ibm-1399
ibm-16684_P110-2003	ibm-16684
ibm-16804_X110-1999	ebcdic-ar
ibm-16804_X110-1999,swaplfnl	ibm-16804-s390
ibm-273_P100-1995	ebcdic-de
ibm-277_P100-1995	EBCDIC-CP-DK/NO
ibm-278_P100-1995	ebcdic-cp-fi/se/sv
ibm-280_P100-1995	ebcdic-cp-it
ibm-284_P100-1995	ebcdic-cp-es
ibm-285_P100-1995	ebcdic-cp-gb
ibm-290_P100-1995	EBCDIC-JP-kana
ibm-297_P100-1995	ebcdic-cp-fr
ibm-33722_P120-1999	ibm-33722-P120-1999
ibm-33722_P12A-1999	EUC-JP
ibm-367_P100-1995	ibm-367_P100-1995
ibm-37_P100-1995	ibm-037 (ebcdic-cp-us/ca/wt/nl)
ibm-37_P100-1995,swaplfnl	ibm-37-s390
ibm-420_X120-1999	ebcdic-cp-ar1
ibm-424_P100-1995	ebcdic-cp-he
ibm-437_P100-1995	ibm-437
ibm-4899_P100-1998	ibm-4899
ibm-4909_P100-1999	ibm-4909
ibm-4971_P100-1999	ibm-4971
ibm-500_P100-1995	ebcdic-cp-be/ch
ibm-5123_P100-1999	ibm-5123
ibm-5346_P100-1998	ibm-5346
ibm-5347_P100-1998	ibm-5347
ibm-5348_P100-1997	ibm-5348
ibm-5349_P100-1998	ibm-5349
ibm-5350_P100-1998	ibm-5350
ibm-5351_P100-1998	ibm-5351

Character set code	Data language
ibm-5352_P100-1998	ibm-5352
ibm-5353_P100-1998	ibm-5353
ibm-5354_P100-1998	ibm-5354
ibm-5478_P100-1995	GB_2312-80
ibm-737_P100-1997	ibm-737
ibm-775_P100-1996	ibm-775
ibm-803_P100-1999	ebcdic-803
ibm-813_P100-1995	Greek8
ibm-838_P100-1995	IBM-Thai
ibm-8482_P100-1999	ibm-8482
ibm-850_P100-1995	ibm-850
ibm-851_P100-1995	ibm-851
ibm-852_P100-1995	ibm-852
ibm-855_P100-1995	ibm-855
ibm-856_P100-1995	ibm-856
ibm-857_P100-1995	ibm-857
ibm-858_P100-1997	ibm-858
ibm-860_P100-1995	ibm-860
ibm-861_P100-1995	ibm-861
ibm-862_P100-1995	ibm-862
ibm-863_P100-1995	ibm-863
ibm-864_X110-1999	ibm-864
ibm-865_P100-1995	ibm-865
ibm-866_P100-1995	ibm-866
ibm-867_P100-1998	ibm-867
ibm-868_P100-1995	ibm-868
ibm-869_P100-1995	ibm-869
ibm-870_P100-1995	ebcdic-cp.roece/yu
ibm-871_P100-1995	ebcdic-is
ibm-874_P100-1995	ibm-874
ibm-875_P100-1995	ibm-875
ibm-878_P100-1996	KOI8-R
ibm-897_P100-1995	ibm-897
ibm-9005_X100-2005	ibm-9005_X100-2005
ibm-901_P100-1999	ibm-901
ibm-902_P100-1999	ibm-902
ibm-912_P100-1995	Latin2
ibm-913_P100-2000	Latin3
ibm-914_P100-1995	Latin4
ibm-915_P100-1995	Cyrillic
ibm-916_P100-1995	Hebrew

Character set code	Data language
ibm-918_P100-1995	ebcdic-cp-ar2
ibm-920_P100-1995	Latin5
ibm-921_P100-1995	ibm-921
ibm-922_P100-1999	ibm-922
ibm-923_P100-1998	Latin-9
ibm-930_P120-1999	ibm-930
ibm-933_P110-1995	ibm-933
ibm-935_P110-1999	ibm-935
ibm-937_P110-1999	ibm-937
ibm-939_P120-1999	ibm-939
ibm-942_P12A-1999	shift_jis78
ibm-943_P130-1999	Shift_JIS
ibm-943_P15A-2003	MS_KANJI
ibm-9447_P100-2002	ibm-9447
ibm-9449_P100-2002	ibm-9449
ibm-949_P110-1999	ibm-949_P100-1999
ibm-949_P11A-1999	ibm-949_P11A-1999
ibm-950_P110-1999	ibm-950_P110-1999
ibm-954_P101-2000	ibm-954_P101-2000
ibm-964_P110-1999	EUC-TW
ibm-970_P110-1995	ibm-eucKR
ibm-971_P100-1995	ibm-971_P100-1995
IBMKANJI	IBM Kanji
IMAP-mailbox-name	IMAP-mailbox-name
ISCII,version=0	x-iscii-de
ISCII,version=1	x-iscii-be
ISCII,version=2	x-iscii-pa
ISCII,version=3	x-iscii-gu
ISCII,version=4	x-iscii-or
ISCII,version=5	x-iscii-ita
ISCII,version=6	x-iscii-te
ISCII,version=7	x-iscii-ka
ISCII,version=8	x-iscii-ma
ISO_2022,locale=ja,version=0	ISO 2022, locale=ja,version=0
ISO_2022,locale=ja,version=1	ISO 2022, JIS, locale=ja,version=1
ISO_2022,locale=ja,version=2	ISO 2022, locale=ja,version=2
ISO_2022,locale=ja,version=3	ISO 2022, JIS7,locale=ja,version=3
ISO_2022,locale=ja,version=4	ISO 2022, JIS8,locale=ja,version=4
ISO_2022,locale=ko,version=0	ISO 2022,locale=ko,version=0
ISO_2022,locale=ko,version=1	ISO 2022,locale=ko,version=1
ISO_2022,locale=zh,version=0	ISO 2022,locale=zh,version=0

Character set code	Data language
ISO_2022,locale=zh,version=1	ISO_2022,locale-zh,version=1
ISO-8859-1	Latin1
Latin1	Latin1
LMBCS-1	LMBCS-1
LMBCS-11	LMBCS-11
LMBCS-16	LMBCS-16
LMBCS-17	LMBCS-17
LMBCS-18	LMBCS-18
LMBCS-19	LMBCS-19
LMBCS-2	LMBCS-2
LMBCS-3	LMBCS-3
LMBCS-4	LMBCS-4
LMBCS-5	LMBCS-5
LMBCS-6	LMBCS-6
LMBCS-8	LMBCS-8
macos-0_2-10.2	macintosh
macos-2566-10.2	Big5-HKSCS (macos)
macos-29-10.2	x-mac-ce
macos-35-10.2	x-mac-turkish
macos-6-10.2	x-mac-greek
macos-7_3-10.2	x-mac-cyrillic
MIXED	Contains items/syntax of multiple data languages
Native	*Native
SCSU	SCSU
SJIS	SJIS
UNICODE_BE	Unicode Big Endian
UNICODE_LE	Unicode Little Endian
US-ASCII	US_ASCII
UTF-16	UTF-16
UTF16_OppositeEndian	UTF16 Opposite Endian
UTF16_PlatformEndian	UTF16 Platform Endian
UTF-16BE	UTF-16 Big Endian
UTF-16LE	UTF-16 Little Endian
UTF-32	UTF-32
UTF32_OppositeEndian	UTF32 Oppostie Endian
UTF32_PlatformEndian	UTF32 Platform Endian
UTF-32BE	UTF-32 Big Endian
UTF-32LE	UTF32 Little Endian
UTF-7	UTF7
UTF-8	UTF-8

Character set code	Data language
UTF-8	UTF-8
windows-1256-2000	Windows-1256
windows-874-2000	Windows 874
windows-936-2000	GBK
windows-949-2000	Windows 949 (Korean)
windows-950-2000	BIG5

*National language is Western or Japanese.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
577 Airport Blvd., Suite 800
Burlingame, CA 94010
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming interface information

Programming interface information, if provided, is intended to help you create application software using this program.

General-use programming interfaces allow you to write application software that obtain the services of this program's tools.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification and tuning information is provided to help you debug your application software.

Warning: Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

Trademarks and service marks

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States or other countries, or both:

AIX
AIX 5L
AS/400
Ascential
Ascential DataStage
Ascential Enterprise Integration Suite
Ascential QualityStage
Ascential RTI
Ascential Software
Ascential
CICS
DataStage
DB2
DB2 Universal Database
developerWorks
Footprint
Hiperspace
IBM
the IBM logo
ibm.com
IMS
Informix
Lotus
Lotus Notes
MQSeries
MVS
OS/390
OS/400
Passport Advantage
Redbooks
RISC System/6000
Roma
S/390
System z
Trading Partner
Tivoli

WebSphere
z/Architecture
z/OS
zSeries

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

This product includes software developed by the Eclipse Project (http://www.eclipse.org/).

IBM WebSphere Transformation Extender, Version 8.1

Index

Special characters

$
  in a component rule 6
[ ] reserved symbol 4
@ reserved symbol 5
< > reserved symbol 3

A

ABS function 143
ABSENT function 95
ACOSINE function 143
ADDDAYS function 63
ADDHOURS function 64
ALL function 107
arguments
  of a function 12
arithmetic operators 11
ASIN function 143
ATAN function 144
ATAN2 function 144

B

BCDTOHEX function 35
BCDTOINT function 36
BCDTOTEXT function 37, 153
bound objects
  in an expression 9

C

card name
  in an expression 3
CHOOSE function 119
CLONE function 27
code pages 201
colon
  in a component path 4
comment name 5
comparison operators 11
component
  in a component list 4
  in an expression 4
component path 3, 4
CONTAINSERRORS function 75, 95
CONVERT function 38
COSINE function 144
COSINEH function 144
COUNT function 144
COUNTABS function 145
COUNTSTRING function 153
CSERIESTOTEXT function 201
CTEXT function 156
CURRENTDATE function 65
CURRENTDATETIME function 66
CURRENTTIME function 67

D

DATETONUMBER function 38, 68
DATETOTEXT function 39, 68, 156
DBLOOKUP function 81, 119
DBQUERY function 83, 122
DDEQUERY function 86, 125
DEFAULT function 27

E

ECHOIN function 28
EITHER function 107
evaluation
  of component rules 6
  of expressions 6
  of functional maps 15
  of functions 12
  of map rules 1, 6
    influence of card order 7
    influence of functions 8
    influence of object names 9
  of operands 12
EXIT function 87
EXP function 146
expressions
  bound objects in 6
  definition of 1
  evaluating 6
  functional map names in 15
  operators in 10
  shorthand notation in 6
EXTRACT function 126

F

FACTORIAL function 146
FAIL function 75
FILLLEFT function 157
FILLRIGHT function 158
FIND function 159
floating component type
  in an expression 5
FROMBASETEN function 40, 146
FROMDATETIME function 41, 69
FROMNUMBER function 41
Function
  used to convert output item value 18
functional maps
  evaluation of 15
  name in an expression 15
functions
  arguments 12
  creating custom 185
  custom 185
  evaluation of 12

G

GETANDSET function 126
GETDIRECTORY function 127
GETFILENAME function 129
GETLOCALE function 128
GETRESOURCEALIAS function 130
GETRESOURCENAME function 131
GETTXINSTALLDIRECTORY function 131

H

HANDLEIN function 28
HEXTEXTTOSTREAM function 42, 160

I

IF function 108
IN
  in component path 5
  reserved word 2, 10
INDEX function 132
INDEXABS function 133
indexing
  an input 4
input
  indexing 4
INT function 43, 147
ISALPHA function 96, 109
ISERROR function 76, 97
ISLOWER function 97, 110
ISNUMBER function 98, 110
ISUPPER function 99, 111

L

LAST reserved word 4
LASTERRORCODE function 133
LASTERRORMSG function 134
LEAVEALPHA function 161
LEAVEALPHANUM function 161
LEAVENUM function 162
LEAVEPRINT function 162
LEFT function 163
literals
  definition of 1
  in expressions 1
local type name 3
LOG function 147
LOG10 function 147
logical operators 11
LOOKUP function 135
LOWERCASE function 164

M

map names in an expression 15
map rules
  evaluation 1
mathlib functions
  ACOSINE 143
  ASIN 143
  ATAN 144
  ATAN2 144
  COSINE 144
  COSINEH 144
mathlib functions *(continued)*
  EXP 146
  FACTORIAL 146
  LOG 147
  LOG10 147
  POWER 149
  RAND 150
  SIN 150
  SINH 150
MAX function 70, 148, 164
MEMBER function 99, 136
MID function 165
MIN function 70, 148, 165
MOD function 149

N

names
  data object 2
  in a component rule 3
  in a map rule 2
NONE reserved word 16
  assigned to an output number 18
  when an input argument of a function evaluates to 16
  when an input argument of a functional map evaluates to 17
  when an operand evaluates to 16
numbers
  as literal 1
NUMBERTODATE function 44, 71
NUMBERTOTEXT function 44, 166

O

object names 2
OFFSET function 101
ONERROR function 77
operands
  evaluation of 12
operators
  arithmetic 11
  comparison 11
  in expressions 10
  logical 11
  order of evaluation 11
output items
  conversion of 18

P

PACK function 45
PACKAGE function 46, 167
PARTITION function 102
partitions
  in an expression 3
POWER function 149
PRESENT function 102
PUT function 90

R

RAND function 150
REFORMAT function 30
REJECT function 78
reserved symbol 4

reserved words and symbols 4, 6
$ 6
[ ] 4
@ 5
< > 3
COMPONENT 3
LAST 4
resourcelib functions
GETLOCALE 128
GETRESOURCEALIAS 130
GETTXINSTALLDIRECTORY 131
REVERSEBYTE function 168
RIGHT function 168
ROUND function 150
RUN function 91, 197

S

SEARCHDOWN function 137
SEARCHUP function 138
SERIESTOTEXT function 47, 169
SETOFF function 31
SETON function 31
shorthand notation 6
in a map rule 6
SIN function 150
SINH function 150
SIZE function 103
SORTDOWN function 139
SORTUP function 140
SQRT function 151
STREAMTOHEXTEXT function 48
SUM function 151
SYMBOL function 49

T

TESTOFF function 32, 104
TESTON function 33, 104
text
as a literal 1
TEXT function 172
TEXTTOBCD function 50, 173
TEXTTODATE function 51, 72
TIMETOTEXT function 73, 175
TOBASETEN function 54
TODATETIME function 55, 74, 176
TONUMBER function 56, 176
TRIMLEFT function 178
TRIMRIGHT function 179
TRUNCATE function 152
type names
local 3

U

UNIQUE function 140
UNPACK function 58
UNZONE function 58
UPPERCASE function 180

V

VALID function 79, 105
VALIDATE function 183

W

WHEN function 113
WORD function 180

X

xmllib functions
VALIDATE 183
XPATH 183
XSLT 184
XPATH function 183
XSLT function 184

Z

ZONE function 59

Printed in USA