Data Integrator

Samples

Handbook

Pervasive Software, Inc.
12365-B Riata Trace Parkway
Austin, Texas 78727 USA

Telephone: 888.296.5969 or 512.231.6000
Fax: 512.231.6010
Email: info@pervasiveintegration.com
Web: http://www.pervasiveintegration.com

PERVASIVE®

See copyrights.txt in the product installation directory for information on third-party and open source software components.

Samples Handbook
October 2010

Contents

About the Samples v
Preparing to Use the Samples vi
Finding Useful Samples vii
Skill and Experience Requirements ix
Programming Samples xi
Command Line Continuation xii

Basic Samples
1 Using Macros to Change Connections 1-1
2 Using Conditional Put Actions with Event Handlers 2-1
3 Filtering Source Data. 3-1
4 Sorting Source Data 4-1
5 Standardizing Multiple Date Formats. 5-1
6 Using DJX to Pass Variables to a SQL Stored Procedure 6-1

Intermediate Samples
7 Using an Exe Step to Run Multiple Transformations 7-1
8 Using Global Variables in Transformations 8-1
9 Using the FileList Function in a Process 9-1
10 Mapping Database Records to EDI 10-1
11 Setting OnDataChange Events 11-1
12 Using Buffered Put Tree to Create Hierarchical Records 12-1

Advanced Samples
13 Aggregating Records 13-1
14 Manipulating Binary Dates at the Bit Level 14-1
15 Complex Date Filtering 15-1
16 Working with DJRowSet and Arrays 16-1
17 Dynamic SQL Lookup 17-1
18 Dynamic SQL Lookup with Error Handling 18-1
19 Dynamic SQL Lookup with Reject Records Handling. 19-1
20 Using EDI X12 Iterator to Read Messages 20-1

Connector-Specific Samples
22 Microsoft Dynamics GP 10: Updating Records 22-1
23 Microsoft Dynamics CRM 4.0: Inserting Records 23-1
24 Oracle Siebel CRM On Demand 14: Deleting Child Records 24-1
25 Netsuite 2.6: Entering Sales Orders 25-1
26 Netsuite 2.6: Adding Addresses to Addressbook 26-1
27 Converting an EDI Source Containing an N1 Loop 27-1

About the Samples

This documentation leads you through transformation and process samples illustrating integration tasks and workflow. The samples are arranged according to user experience, from basic to advanced. Before reviewing the samples, see the following sections:

- "Preparing to Use the Samples" on page -vi
- "Skill and Experience Requirements" on page -ix
- "Programming Samples" on page -xi
- "Command Line Continuation" on page -xii

The following samples document connector-specific features new to the Data Integrator 9.2 release.

- "Microsoft Dynamics GP 10: Updating Records" on page 22-1
- "Microsoft Dynamics CRM 4.0: Inserting Records" on page 23-1
- "Oracle Siebel CRM On Demand 14: Deleting Child Records" on page 24-1

Contact Us

If you have questions about a particular sample, or if you want to request that a new sample be created, please contact samplesinfo@pervasive.com.

Preparing to Use the Samples

Samples offer a different way of learning from tutorial procedures. Once you understand the samples offered here, you can copy and adapt them to your own situation.

In recent product releases, these samples were delivered in an .msi installer. Starting with Data Integrator 9.2.0, the samples are available for download from the Pervasive web site.

Tip Please note that some of the transformations and processes are not documented yet. Check back often to download new sample files and to view new online documentation.

Before working with the samples, you perform the following steps:

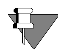
1. Download the Sample Files
2. Define a Samples Repository
3. Set Map All View as Default

Download the Sample Files

➢ To download the samples

1. Download the Product_Samples.zip file.
2. Unzip the Product_Samples.zip file to the following location on your computer:

   <drive>:\Documents and Settings\All Users\Product Samples



Note Please note that if you use a different sample location than the suggested path above, you must change all paths in each sample.

In this document we refer to this path as *SamplesDir*.

Define a Samples Repository

The sample files work best in their own repository.

➢ **To set up samples in a repository**

1. In Repository Explorer, select **File** > **Manage Repositories**.
2. In the **Select Repository** dialog box, click **Add**.
3. Enter a name for the sample repository, such as Product Samples, and click **Find**.

   Enter your *SamplesDir* path here.
4. Click **OK** to save the new repository.

   The sample files are now available to all integration applications.

Set Map All View as Default

Map Designer has two Map tabs: the Map Fields tab and Map All tab. Before working with the samples, set the Map All tab as the default in Map Designer. The Map All tab displays the information needed for the samples to be most easily understood.

➢ **To set Map All tab as default**

1. Open Map Designer and select **View** > **Preferences**, then click the **General** tab.
2. Select the check box **Always show Map All view**, which displays a navigation tree with events.

Finding Useful Samples

The samples in this document are grouped by complexity as basic, intermediate, and advanced. Samples related to specific connectors are also listed under their connector name.

Each sample lists the tools used and what was done to generate the target sample results.

If you are just beginning to work with the integration tools, you may want to work through the entire series of samples to learn to perform tasks from basic to an advanced level.

Use the index to find samples based on source and target file types (such as binary, ASCII, or dBASE), scripting expressions, and particular techniques, such as use of structured schemas and declaring of variables.

Note Components are occasionally renamed, so you may see an earlier version mentioned in an older sample. For information on updated component names, see the release notes.

Skill and Experience Requirements

Before reviewing and trying to emulate any samples from this document, you should be aware of the skill level needed to perform those tasks. For example, if you are totally new to the integration platform, you may not be ready to create a transformation that requires use of a database or a scripting language. By the same token, reviewing a sample transformation without the prerequisite knowledge or experience may cause more confusion than clarification.

Each sample gives its skill level. Before using a sample, note this level to ensure that the sample is appropriate for your skills. The following describes the basis for basic, intermediate, and advanced levels.

Basic

- User is still fresh from learning concepts such as schema, source, target, event, transformation, component types, and so forth
- Has basic knowledge to run integration products using their GUIs and tends to prefer them to working from the command-line interface
- Uses functions and simple flow control constructs for scripting;
- When learning a new concept, prefers being shown only one way to achieve a goal
- Starting to use the simplest samples and use cases
- Not strong at troubleshooting; needs more assistance in resolving issues
- Can get lost fairly easily when performing complex tasks

Intermediate

- User is familiar with integration concepts and terms
- Familiar with event handlers and actions
- Familiar with RIFL functions and objects
- When learning a new concept, likes to consider more than one way to perform a task
- Sometimes works from the command-line interface
- Can revise existing scripts and occasionally writes new scripts for more complex data manipulation

- Can benefit from more elaborate samples and use cases
- Has learned to do general troubleshooting

Advanced

- User at this level is frequently a developer
- Comfortable with integration concepts and terms, event handlers and actions, and RIFL functions and objects
- May use SDKs and embed Integration Engine in other products
- Often works from the command-line interface
- Experienced at scripting and user-defined functions
- Big consumer of samples and use cases, including special cases and customizations
- Knows many troubleshooting tips and tricks
- Might use Message Component Framework (MCF) to develop custom components

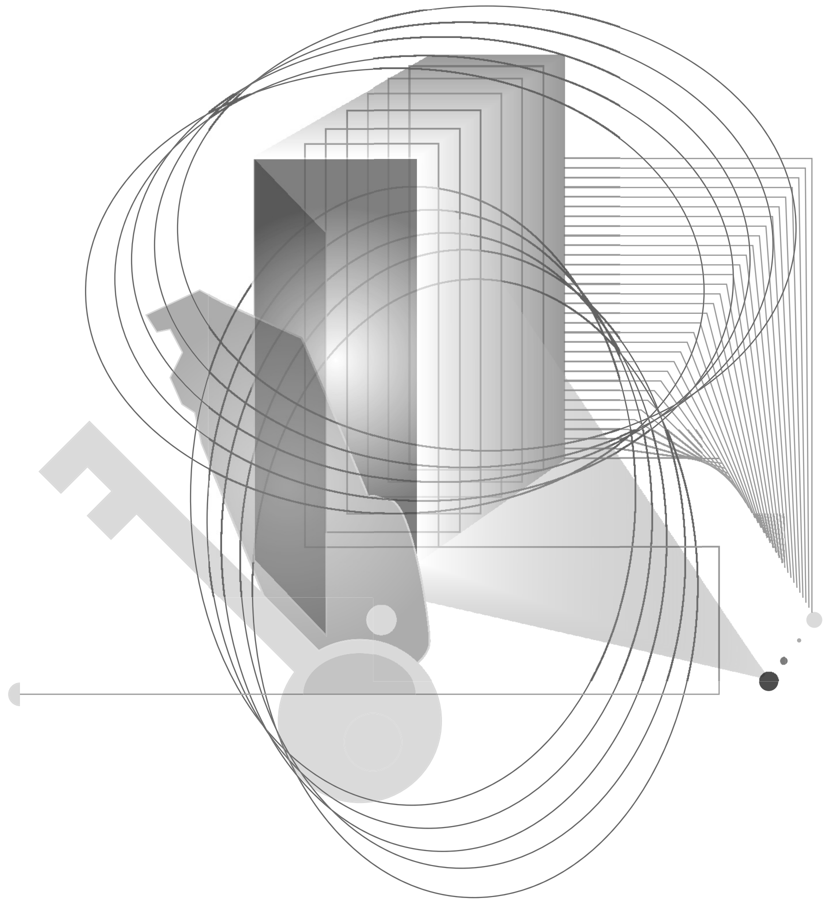
In addition to understanding the skill level needed to perform the steps, certain skill sets and specific experience may also be needed to effectively work with a sample. For example, most samples require that you have experience using Map Designer because that is the basic tool for transformation creation and execution. If the sample requires that you create or understand the RIFL scripting language, that requirement is also listed. And finally, if other tools and experience are needed to run a sample (such as a third-party database), those requirements are also listed.

Programming Samples

The samples documented here deal with the design-time graphical user interface and integration applications. Additional samples, including command-line interface samples, are available in the *Integration Engine SDK Getting Started Guide*. Refer to that manual for more samples that can be copied and customized to your needs.

Command Line Continuation

This documentation includes many command-line and scripting examples. If a procedure calls for the command to be entered on one line, this is noted in the procedure. Many of the command-line examples wrap to the next line. To present clearer examples, command-line continuation characters are not used.



Basic Samples

Using Macros to Change Connections

1

Changing From One Life Cycle Environment to Another With Macros

Macros are symbolic names assigned to text strings, usually in file paths. You can use macros as a tool as you move integration project files from one life cycle environment to the next.

Important Note

This sample transformation is not designed to be run. It shows how to use macros to connect to source and target files, tables, or entities in your own life cycle environments.

Objectives

This transformation demonstrates how to use macros to change from development, test, and production environments.

Skill Level

Basic

Skill Set and Experience

- Map Designer

Sample Map Location

SamplesDir`\Transformations\`macros_switching_environments.map.xml

Sample Repository Configuration

The samples use workspaces and repositories to access the sample files. You must define that repository before running a sample transformation or process. For more information, see "Define a Samples Repository" in the "About the Samples" section.

Source Information

Source Connector: Excel XP
Source File: Macro points to Excel spreadsheet

Target Information

Target Connector: SQL Server 2005
Output Mode: Update
Target File/URI: Macro points to SQL Server table

Design Review

➢ To define macros for source, target connections

1. First, we selected a Source connector for the transformation.
2. Next, we selected **Tools** > **Define Macros** and selected **New**. Then we typed the macro name, value, and description.

   Name: ACCOUNTS

   Value: C:\Documents and Settings\All Users\Product_Samples\Data\AccountSummariesbyState.xls

   Description: (optional) Macro that points to AccountSummariesbyState.xls file.
3. Then we returned to the Source tab at **File** and selected **Tools** > **Paste Macro String**. The macro string $(ACCOUNTS) appears.
4. We repeated the same procedure for the target connection information. As shown on the Target Connection tab, you can use macros to store Server $(ServerPath), User ID $(MyUserID), and Password information.

Tip Note that macros can point to any of your development, testing, or production environments. To change environments, select **Tools** > **Define Macros**, select the macro, and choose **Edit**. At Macro Value, enter the new environment location. For instance, to move from test to production, change the path \\testserver\testdata to \\productionserver\data.

Reference

For information on basic macro usage and syntax, see “Macro Manager” in the *Getting Started Guide*.

For information on using life cycle environments, see “Moving from Development to Test and Production Environments” in the *Best Practices Handbook*.

To learn more about using macros to change from one life cycle environment to another, see “Using Macros During Deployment” in the *Best Practices Handbook*.

Using Conditional Put Actions with Event Handlers 2

In this sample we examine the use of event handler actions that allow Map Designer to process record data on a conditional basis.

Objectives

Use a conditional put record action to evaluate the record and act accordingly. If the source date is in a valid format, the record should be written to the target file. If the source date is invalid, the record **should not** be written to the target and an error message should display.

Skill Level

Basic

Skill Set and Experience

- Map Designer
- Basic RIFL Scripting

Sample Map Location

SamplesDir`\Transformations\Events_ConditionalPut.map.xml`

Sample Repository Configuration

The samples use workspaces and repositories to access the sample files. You must define that repository before running a sample transformation or process. For more information, see "Define a Samples Repository" in the "About the Samples" section.

Source Information

Connection
Source Connector: **ASCII (Delimited)**
Source File/URI:
SamplesDir`\Data\Accounts.txt`

Description
The source file is a simple text file containing over 200 records. In this sample we are concentrating on the **Birth Date** column.

Target Information

Connection
Target Connector: **ASCII (Fixed)**

Output Mode: **Replace File/Table**
Target File/URI:
SamplesDir`\Data\Accounts_Out.txt`

Description
The target is a text file that resembles the source file except that it has fewer records, since those with invalid dates in the source file were not written to the target file.

Procedure Explanation

The following steps describe the script used in the **AfterEveryRecord** event handler on the source **R1** table. You can view this script by opening the **count** parameter on the **ClearPutMap Record** action.

1 First we declared a variable of **A** and defined it to represent the source **Birth Date** field value:

```
Dim A
'Convert the string into a date.
A = Records("R1").Fields("Birth Date")
```

2 Next we started the condition definition. If a record matches the function definition (in this case the date test **IsDate**), then it is considered *true*. We set the condition to resolve to the number **1** if the response is true:

```
if IsDate(A) Then
  ' Enable the Put action by setting to one
1
```

3 If the condition is not true (date not in correct format), we set the condition to write an error message to the log, increment the error counter (**myBadDates**), and discard the bad record:

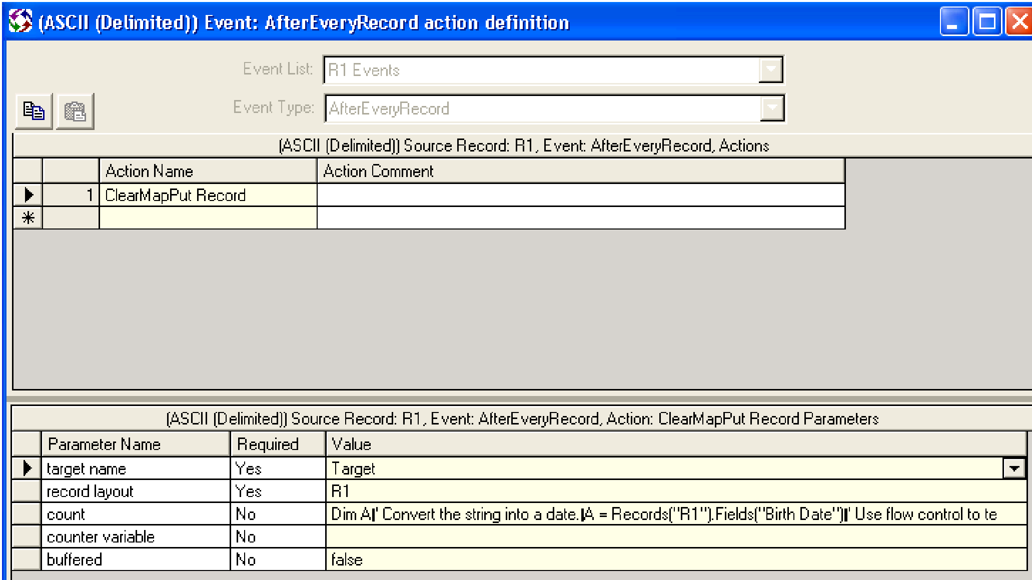
```
Else
  ' Invalid date, log a message
  Logmessage("Error", "Account number: " &
  Records("R1").Fields("Account Number") & _
  " Invalid date: " & Records("R1").Fields("Birth
Date"))
  ' Increment counter
  myBadDates = myBadDates + 1
  ' Suppress the Put action by setting to zero
0
```

4 We end the **if** condition:

```
End if
```

More Detailed Information

Some actions can be fired based on condition. Those actions will have **count** and **count variable** parameters. The **count** parameter accepts any expression that returns a number value. When this value is *zero* (**0**), the action is not performed, as in our date example. When the value is *one* (**1**), the action is performed. When the value is *greater than one*, the action is performed that many times (with the **counter variable** parameter providing an index).



Reference

See “Event Actions” and “Event Handling” in the *Intermediate and Advanced Mapping User’s Guide.*

Also see “If...Then...Else Statement” in the *Rapid Integration Flow Language Reference.*

Filtering Source Data

3

You can restrict the records written to target files in several ways. This sample illustrates the quick and easy Map Designer filter utility.

Objectives

Select only the account records with a Texas address (**TX** in the **State** field), write those records to the target file, and discard all other source records.

Skill Level

Basic

Skill Set and Experience

- Map Designer
- Basic RIFL Scripting

Design Considerations

You can filter data during source processing, filter processing, or both. The most efficient method for your transformation depends on what you are trying to accomplish. If you filter source records, any records that do not meet your specified criteria are discarded before target data processing takes place. Conversely, when filtering takes place on target data only, all source records are passed directly to the target for processing. If you prefer, you can also use both methods to first do a rough filtering of source data and then perform a secondary filtering of target data for a different criterion.

Sample Map Location

SamplesDir`\Transformations\SourceDataFeatures_Filter.map.xml`

Sample Repository Configuration

The samples use workspaces and repositories to access the sample files. You must define that repository before running a sample transformation or process. For more information, see “Define a Samples Repository” in the “About the Samples” section.

Source Information

Connection
Source Connector: **ASCII (Delimited)**

Source File/URI:
SamplesDir`\Data\Accounts.txt`

Description
The source file is a simple text file containing 206 records. In this sample we are concentrating on the **State** column.

Target Information

Connection
Target Connector: **ASCII (Delimited)**
Output Mode: **Replace File/Table**
Target File/URI:
SamplesDir`\Data\AccountsinTX.txt`

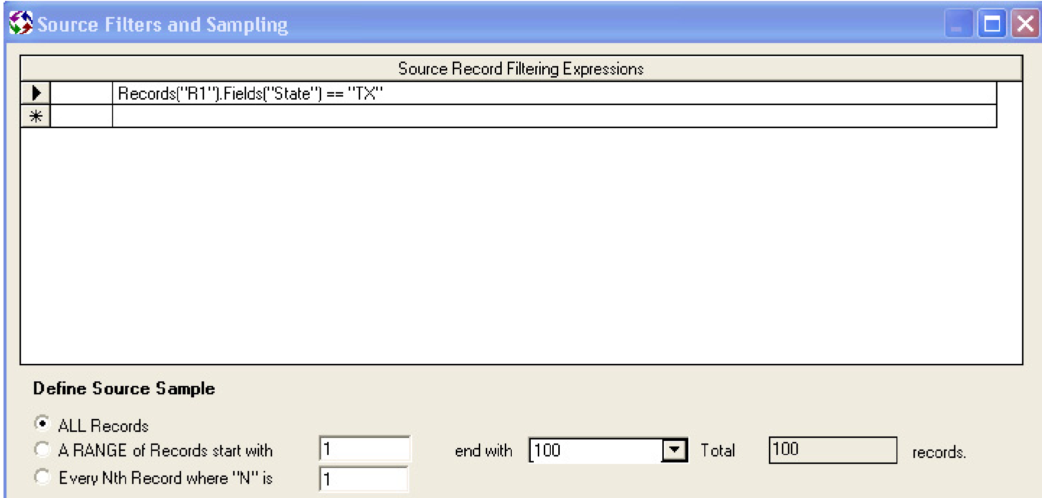
Description
The target is a text file that resembles the source file except that it has fewer records, since only those with those with **TX** in the **State** column are written to the target file.

Procedure Explanation

In this transformation we are filtering out every account where the address is outside the state of Texas.

1. We set up both connections for delimited ASCII files.
2. On the **Map** tab, we drug all records from the source to the target grid.
3. We added a single source **AfterEveryRecord** event handler with a **ClearMapPut Record** action to process and write the records to the target file.

   We clicked the **Source Filters** icon and entered our filter expression in the first row of the **Source Record Filtering Expression** box:



By entering this expression:

```
Records("R1").Fields("State") == "TX"
```

we told Map Designer to process the records that evaluate as **True** (**State** equals **TX** exactly).

Note At the bottom of this window we selected **All Records** to be processed because this is a small source file. However, when processing a larger file with many hundreds or thousands of records, it might be more appropriate to select another value to limit the number of records for processing.

4 After validating the map, we ran the transformation, generating a target file containing 10 account records, all in Texas.

More Detailed Information

You may get unexpected results when specifying both source and target filters for a transformation. For example, if you filter a 5000-record source file to process only the first 1000 records, and then you supply a target filter to write every tenth record to the target, you will get only 100 target records, not 500. Remember that the target filter is applied only to the records that make it through the source filter.

Reference

See "== Operator" in the *Rapid Integration Flow Language Reference*.

Sorting Source Data

4

Most transformations run faster when the data is already sorted into a certain order. Here we review Map Designer sort functionality for sequencing one or more data fields.

Objectives

Sort the account records from a source file and write them to a target file, grouped together by state name in ascending order.

Skill Level

Basic

Skill Set and Experience

Map Designer

Design Considerations

While sorting has overhead associated with it, this process can be essential when the source is in text format and cannot otherwise be accessed in a specific sequence. If the time required for sorting data becomes a major factor, you may need to employ other strategies. On the other hand, the benefits gained from working from a source file in the proper sequence may be greater than the time expenditure.

Another consideration is if any other processing will be performed on the data. For example, the source input must be in the proper sequence to use **OnDataChange** events. See "Setting OnDataChange Events" for a sample of that usage.

Sample Map Location

SamplesDir`\Transformations\SourceDataFeatures_Sort.map.xml`

Sample Repository Configuration

The samples use workspaces and repositories to access the sample files. You must define that repository before running a sample transformation or process. For more information, see "Define a Samples Repository" in the "About the Samples" section.

Source Information

Connection
Source Connector: ASCII (Delimited)

Source File/URI:
SamplesDir`\Data\Accounts.txt`

Description
The source is a simple text file containing 206 records. In this sample we are concentrating on the **State** column.

Target Information

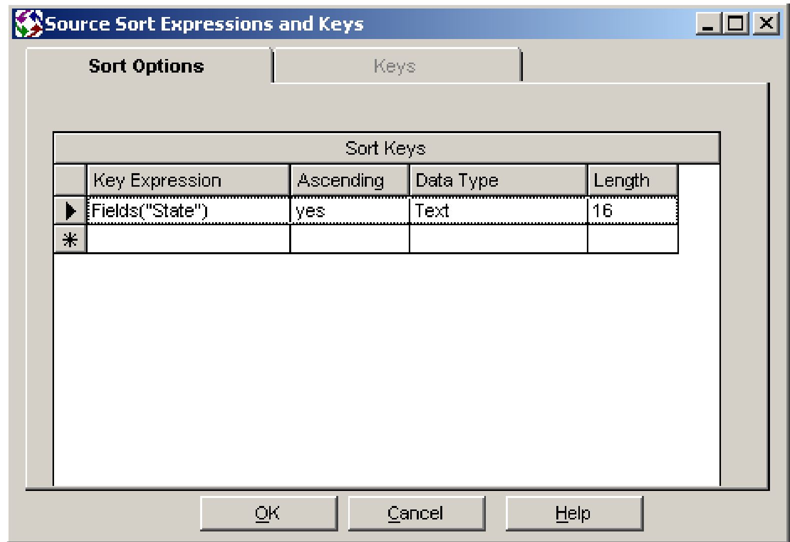
Connection
Target Connector: **ASCII (Delimited)**
Output Mode: **Replace File/Table**
Target File/URI:
SamplesDir`\Data\AccountsSortedbyState.txt`

Description
The target is a text file identical to the source file except that all records are sorted by the name in the **State** column. They are in ascending order, which is the default sort option.

Procedure Explanation

We used the following steps to read the source data, sort it by the value in the **State** field, and write the data to a target file in that sequence.

1. We set up the connections for the source and target files.
2. On the **Map** tab, we drug all fields from the source grid to the target grid (using the same fields in target as in source).
3. We added a single source **AfterEveryRecord** event handler with a **ClearMapPut Record** action to process and write the records to the target file.
4. We clicked the **Source Keys and Sorting** icon in the toolbar to open the **Source Sort Expressions and Keys** window.
5. On the **Sort Options** tab, we clicked the first **Key Expression** field to display the down arrow.
6. We clicked the down arrow and selected the **State** field to use as the sort key.



7 We accepted the default values for the remainder of the row and clicked **OK** to save and exit this window.

8 We validated the map and ran the transformation.

The resulting target file contains the same number of records as the source file, but they are all in ascending state-name sequence.

More Detailed Information

Using this same method, we could have performed a secondary sort of the records by other fields, if necessary. For example, to get a finer granularity of address information, we might need to also sort the records by city. To do that, we just select the second **Key Expression** row on the **Sort Options** tab and select **City** as the next sort key. We can continue in this way for all the fields in the table, if appropriate. The sequence in which the field names appear in this grid determines the sort order of fields.

Reference

See “Sort the Target File” on page 2-14 and “Sorting Data in Append Mode” on page 3-8 in *Map Designer User’s Guide*.

Standardizing Multiple Date Formats

5

This sample transformation demonstrates how to standardize date formats for the target when the source file dates are in different formats.

Objectives

Standardize source dates in different formats.

Skill Level

Basic

Skill Set and Experience

- Map Designer
- RIFL Scripting
- Basic understanding of date formats and the importance of storing dates in a single, standard format

Design Considerations

The principal design consideration is to choose the date format that fits your business process requirements. Following this decision, the task is to write a RIFL expression that converts any date in the source file to the desired target date format.

Sample Repository Configuration

The samples use workspaces and repositories to access the sample files. You must define that repository before running a sample transformation or process. For more information, see "Define a Samples Repository" in the "About the Samples" section.

Sample Map Location

SamplesDir`\Transformations\Standardize Various Date Formats.map.xml`

Source Information

Connection
Source Connector: **ASCII (Delimited)**
Source File/URI:
SamplesDir`\Data\src_standard.txt`

Description
The source is a simple ASCII delimited file with sample dates in various standard and nonstandard formats.

Target Information

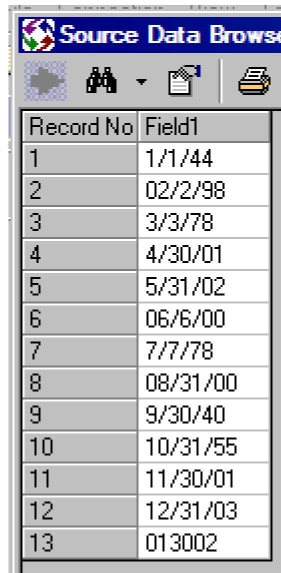
Connection
Target Connector: **ASCII (Delimited)**
Output Mode: **Replace File/Table**
Target File/URI:
SamplesDir`\Data\trg_standard_date_format.asc`

Description
The target is an ASCII delimited file.

Procedure Explanation

1 First, we opened Map Designer and connected to the source and target files.

The sample source file contains only one field, dates in various non-standard formats.



Record No	Field1
1	1/1/44
2	02/2/98
3	3/3/78
4	4/30/01
5	5/31/02
6	06/6/00
7	7/7/78
8	08/31/00
9	9/30/40
10	10/31/55
11	11/30/01
12	12/31/03
13	013002

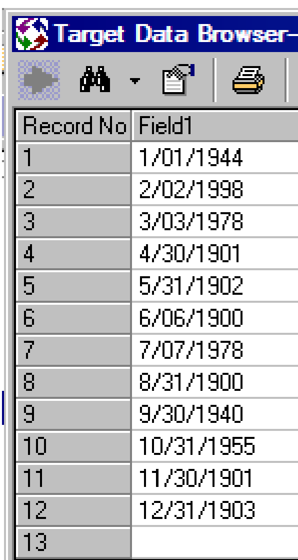
2 After mapping the date field to target at the **Map** tab, we wrote a target field expression that is used for specific date masks and called based on the source date format:

```
a = DateValMask(Fields("Field1"), "m/dd/YY1900")
b = DateValMask(Fields("Field1"), "mm/d/YY1900")
c = DateValMask(Fields("Field1"), "mm/dd/YY1900")
d = DateValMask(Fields("Field1"), "m/d/YY1900")
```

The next expression evaluates the source date and calls the specific **DateValMask** expression to convert the date to the desired format.

```
If Mid(Fields("Field1"), 2, 1) Like "/" And
Mid(Fields("Field1"), 5, 1) Like "/" Then a

ElseIf Mid(Fields("Field1"), 3, 1) Like "/" And
Mid(Fields("Field1"), 5, 1) Like "/" Then b

ElseIf Mid(Fields("Field1"), 3, 1) Like "/" And
Mid(Fields("Field1"), 6, 1) Like "/" Then c

ElseIf Mid(Fields("Field1"), 2, 1) Like "/" And
Mid(Fields("Field1"), 4, 1) Like "/" Then d

End If
```

3. When we ran the transformation, it produced the following target:



Record No	Field1
1	1/01/1944
2	2/02/1998
3	3/03/1978
4	4/30/1901
5	5/31/1902
6	6/06/1900
7	7/07/1978
8	8/31/1900
9	9/30/1940
10	10/31/1955
11	11/30/1901
12	12/31/1903
13	

More Detailed Information

A good understanding of the **DateValMask** function is required. The most important thing to know about the **DateValMask** function is that the mask references the source date format, not the desired target date format. For example, a source date of 7/7/78 is represented by the mask m/d/yy. The field expression to convert this source date is:

```
DateValMask(Fields("sourcedate"), "m/d/YY1900").
```

Reference

See “Picture Masks”, “Converting Dates”, and “DateValMask Function” in the *RIFL Programmer’s Reference Manual*.

process

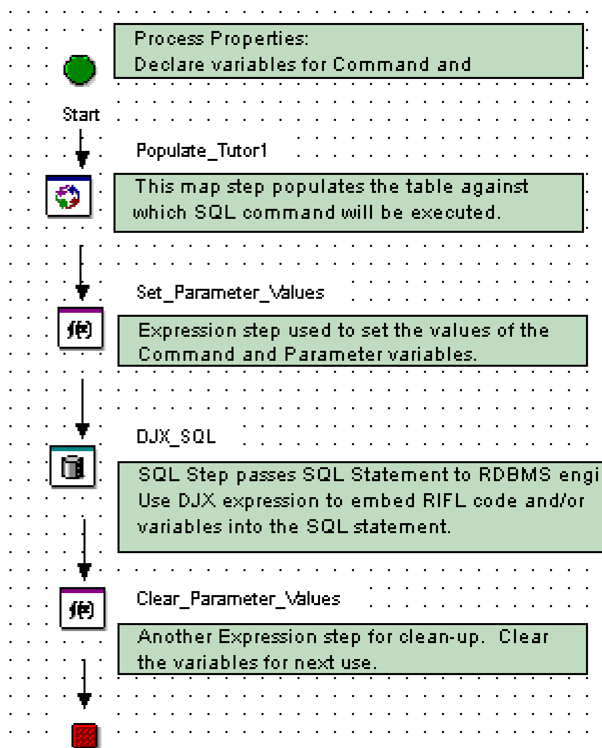
Using DJX to Pass Variables to a SQL Stored Procedure

6

In this sample, we want to to populate an Access 97 database table using Rapid Integration Flow Language (RIFL) and DJX in scripting steps.

Objectives

Using DJX, pass variables to a SQL Stored Procedure and escape into RIFL (Rapid Integration and Flow Language) to design SQL statements. Without the DJX statement, the SQL Statement is treated as a literal SQL statement.



Skill Level

Basic

Skill Set and Experience

- Process Designer
- Map Designer
- RIFL Scripting
- Basic understanding of how to create and use process variables.
- Know how to design a process that includes a transformation step and scripting steps.

Design Considerations

To use Process Designer to run this sample process, you must plan the following in advance:

- *Where is the directory located that holds the files to be processed?* You need this information in order to connect to this directory.

  A major consideration is the location of the target of the transformation in the DJX Stored Procedures process. If the directory resides on a machine other than the one where the process is executed, you must know the path to that machine and have permission to access it.

Sample Process Location

SamplesDir`\Processes\DJX Stored Procedures.ip.xml`

Sample Repository Configuration

The samples use workspaces and repositories to access the sample files. You must define that repository before running a sample transformation or process. For more information, see "Define a Samples Repository" in the "About the Samples" section.

Other Files Location

This process also uses the following transformation file:

SamplesDir`\Transformations\`Populate_Tutor1.map.xml

Procedure Explanation

To create this process in Process Designer, the following was done:

1. **Populate_Tutor1**
   The first process step is a transformation that maps data from an ASCII-delimited source file to an Access 97 target table.

   Tutor1 was not an existing table in the **SamplesWork.mdb** database, so when the transformation step was run for the first time, the target table **Tutor1** was created.

When we saved the transformation and exited Map Designer, Process Designer prompted us to create a SQL Session. We named the Access 97 session **sql**.

2 **Set_Parameter_Values**
The second step uses scripting to set values for the **Command** and **Parameter** variables.

- Before we included scripting steps that use process variables, we created the variables. We did this by selecting **File ▸ Properties**, and then clicked the **Process Variables** tab. We selected **Add** to create new variables and named them as follows:

```
Exec1
Param1
```

  We selected **Variant** as the variable type for both variables.

- Next, we created the scripting step **Set_Parameter_Values** and included the following script:

```
Param1 = "10019"
Exec1 = "Delete from Tutor1 Where [Account No] =
'" & Parm1 & "'"
msgbox(Exec1)
```

3 **DJX_SQL**
This is the SQL step that calls the **sql** session (created in the **Populate_Tutor** step). This step uses the following SQL statement to pass the variable values to the SQL engine:

```
Djx(Exec1)
```

4 **Clear_Parameter_Values**
To clear the parameter values each time the process is run, the following script was added in this step:

```
Parm1 = ""
Exec1 = ""
```

More Detailed Information

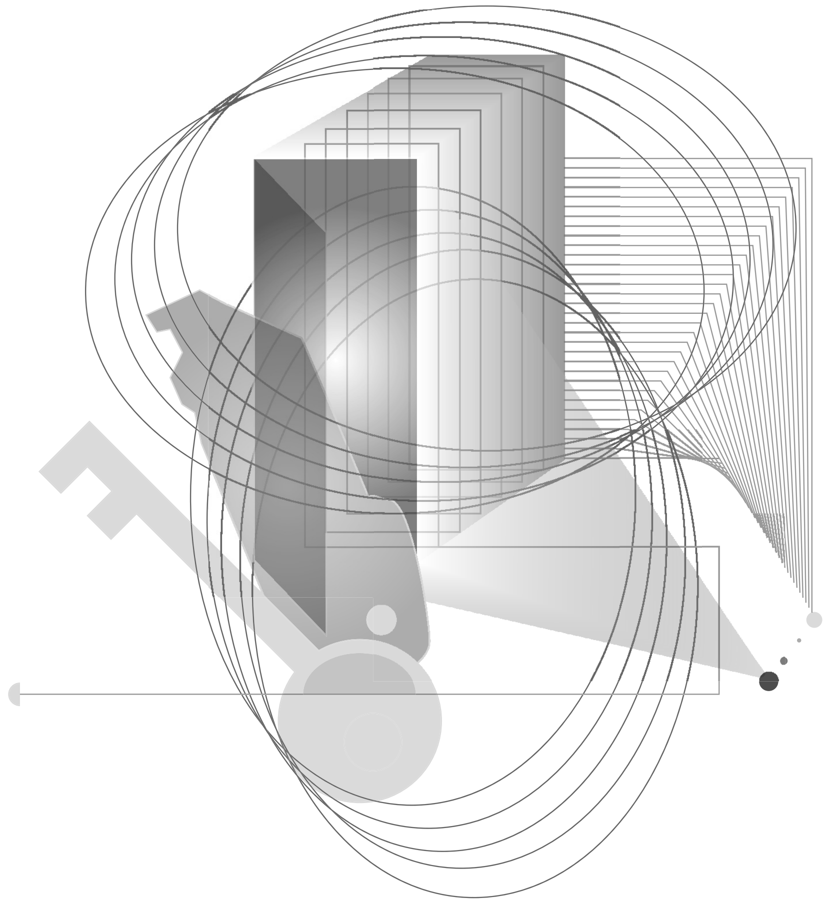
After you have studied the process design, run the process and examine the results. A dialog will display indicating **Delete from Tutor1 Where [Account No] = '10019'**. Click **OK**.

To look at the target table, do the following:

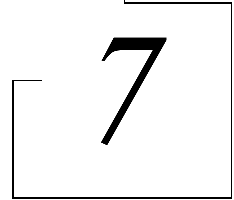
1. Double-click the transformation step to open the transformation dialog.
2. Click **Edit** to open Map Designer.
3. In the main toolbar, click the **Target Data Browser** icon.
4. After the target table opens, notice that the **Account No 10019 record** was deleted from the table.

Reference

See “Using DJX to Create SQL Statements” in the *Rapid Integration Flow Language Reference* available with your product.



Intermediate Samples



Using an Exe Step to Run Multiple Transformations

Objectives

This sample demonstrates running multiple transformations from a process using an Exe step.

Skill Level

Intermediate

Skill Set and Experience

- Process Designer
- Map Designer

Sample Process Location

SamplesDir\Processes\MultipleMapsFromSingleProcess.ip.xml

Sample Repository Configuration

The samples use workspaces and repositories to access the sample files. You must define that repository before running a sample transformation or process. For more information, see “Define a Samples Repository” in the “About the Samples” section.

Before You Begin

➢ To set up process before running

Before you can run the process and view the results, you must do the following tasks:

1. Open Process Designer, then select **Tools** > **Macros** and add a new macro named MultiMapsSingleProcessSample that points to the directory above the one containing the process .ip.xml file.
2. Next, create a macro for each field in *SamplesDir*\Data\TransformationsInfo.mdb. Create the following macro names with some default value. The process overwrites it with values read from the database. Name the macros as follows: mapFilename, srcFilename, trgFilename, logFilename, schemaFilename, and databaseName.

Design Review

The following table provides information at a glance about the four transformations in the process.

Transformation Name	Remarks
AsciiToXMLTutor1	Transforms the Ascii Delimited source file Tutor1.asc into the XML file Tutor1.xml.
ASCIIFixedToHTMLTutor2	Transforms the Ascii Fixed source file Tutor2.asc and the structured schema tutor2.ss.xml into Tutor2.html.
ASCIIToExcelTutor3	Transforms the Ascii Delimited source file Tutor3.asc into Excel file Tutor3.xls.
ExcelToAsciiTutor3	Uses the Excel file generated by the AsciiToExcelTutor3 transformation as the source and transforms it back into Tutor3.txt.

The following table provides information about the process variables set in **File** > **Process Properties**.

Variable Name	Variable Type
Connector	DJImport
DataFilesInfo	DJRowSet
recordCounter	Variant (initial value is 1)

Process Design

Next, we provide details on the process steps.

1. **GetTransformationsInfo step** - Uses a DJImport object to connect to the Access 2000 data connector and read the data in TransformationsInfo.mdb into a DJRowset object.
2. **More Transformations to Run step** - In this Decision step, the process determines whether more transformations exist to run. If true, the next step is CreateMacros, and if false, the process stops.
3. **CreateMacros step** - This Scripting step uses an expression to clear each macro defined in the process. Then it defines macro values and passes them as arguments to the Exe step.

4. **RunTransformations step** - Uses the djengine command to run each transformation. In this case, all the options are specified using macros specified on the command line. The “Start in” field is left blank because we start and run this process from the Process folder that contains the MultipleMapsFromSingleProcess.ip.xml file.
5. **IncrementCounter step** - Increments the recordCounter variable. The process iterates through each record using the recordCounter to determine the EOF (end of file) in the DJRowset object and defines macro values for each field in the record.
6. **More Transformations to Run step** - The process returns to this step to determine if more transformations need to be run. If true, the process repeats, and if false, stops.

Results

Once you run the process, open the TargetFile folder and verify that the Tutor1.xml, Tutor2.html, Tutor3.txt, and Tutor3.xls files are written to the folder. Also open the Logs folder and ensure that a separate log file is written for each target file.

Reuse Notes

Once you have viewed the process sample, you can save the process as a new name in your workspace. Then you can use it as a template each time you want to use a process that runs multiple transformations from an Exe step. Edit the macros to point to each of your process and transformation files.

Reference

See “Using Process Steps to Build Process Designs” in the *Process Designer User's Guide*.

Using Global Variables in Transformations

8

If you need to set variables for use by several different actions in a transformation, global variables provide an easy method for declaring them.

Objectives

Declare a global variable in **Transformation and Map Properties** and use that variable in an event handler action. A successful run of this transformation will result in records with valid dates being written to the target file. Discarded records will be tracked in the message log with an entry identifying those records and the balance from each record added to a running balance.

Skill Level

Intermediate

Skill Set and Experience

- Map Designer
- Basic RIFL Scripting

Design Considerations

One important consideration when defining variables in your transformation is how and where the variable will be used. A variable defined as *public* can be used throughout a project (a set of related transformations and/or processes), while a *private* variable can be used in a single transformation only. Variables declared with a **Dim** expression are specific to a module or an expression. A *global variable* is treated as a *private variable*, unless defined otherwise, and can be used across the entire transformation.

The following table summarizes the scope of variables in our integration language.

	Scope	Availability
Dim	Dim	Local to Script Module
Global	Private	Throughout Map Design
Global	Public	Throughout Process Design

The keyword **Global** has been deprecated and replaced by **Private** and **Public**, depending on your use of a map or a process design.

Sample Map Location

SamplesDir`\Transformations\MapProps_GlobalVariables.map.xml`

Sample Repository Configuration

The samples use workspaces and repositories to access the sample files. You must define that repository before running a sample transformation or process. For more information, see “Define a Samples Repository” in the “About the Samples” section.

Source Information

Connection
Source Connector: **ASCII (Delimited)**
Source File/URI:
SamplesDir`\Data\Accounts.txt`

Description
The source is a simple text file containing 206 records. In this sample we are concentrating on the **Birth Date** column.

Target Information

Connection
Target Connector: **ASCII (Fixed)**
Output Mode: **Replace File/Table**
Target File/URI:
SamplesDir`\Data\Accounts_Out.txt`

Description
The target is a text file that resembles the source file except that it has fewer records, since those in the source with an invalid date are not written to the target.

Procedure Explanation

In this transformation we declared one global variable and used it in one target action.

1. After connecting to the appropriate source and target files, we used drag-and-drop functionality to copy all source fields to the first **Target Field Name** cell.
2. In the source file, all fields were type **Text**. We wanted to perform some calculations on the payment fields, so we changed the target **StandardPayment**, **LastPayments**, and **Balance** fields to type **Decimal**.

3. We selected **View ▸ Transformation and Map Properties ▸ Global Variables** and created a new variable named **varBalance** of data type **Variant**. We left the **Public** checkbox empty so this variable would be treated as private.
4. We added **AfterEveryRecord** as the source R1 record event handler. This event handler uses the **ClearMapPut Record** action to write the record to the target file.
5. We modified the target field expression for **Birth Date** as follows:

   a. We declared a variable as **A** and defined it to represent the **Birth Date** field value:

   ```
   Dim A
   A = Records("R1").Fields("Birth Date")
   ```

   Notice that we use **Dim** to declare the variable **A** because this variable is needed only within the scope of this script module.

   b. We tested variable **A** to determine if the date in the **Birth Date** field is a valid date. If it is valid, it is formatted in the "ddddd" style (using default short style of m/d/yy):

   ```
   If IsDate(A) then
     Format(A,"ddddd")
   ```

   c. At this point we use the **varBalance** global variable to perform some calculations. If the date is not valid, the **varBalance** variable is added to itself to generate a running balance.

   For example, for the first invalid date, **varBalance** (zero at that time) is added to that record's balance (**Balance**), creating a new running balance in the **varBalance** variable. The second invalid date causes the balance for that record to be added to the **varBalance** amount, and so on for each invalid date:

   ```
   Else
     varBalance = varBalance +
     Records("R1").Fields("Balance")
   ```

   d. An entry for the invalid record is also written to the message log, listing the account number, date, and the current running balance:

```
  Logmessage("Warn", "Account Number " &
  Records("R1").Fields("Account Number") & _
  " has an invalid date: " &
  Records("R1").Fields("Birth Date"))
  Logmessage("Info", "  The running balance &
  of all discarded records is " & varBalance)
```

e. Finally we discard the invalid record and end the **If** condition:

```
  Discard()
End If
```

More Detailed Information

The proof of the accuracy of this sample is the target text file and the message log. The source text file contains 206 records before transformation. After running the map, the target text file should contain 201 records because 5 records have invalid dates.

When you view the message log, pay special attention to the following lines:

```
*** Execution Begin: [xmldb:ref://////
MapProps_GlobalVariables.tf.xml]
Account Number 01-032845 has an invalid date: 02/29/1974
The running balance of all discarded records is 239.18
Account Number 01-687977 has an invalid date: 02/31/1956
The running balance of all discarded records is 350.11
Account Number 01-689832 has an invalid date: 02/31/1956
The running balance of all discarded records is 350.11
Account Number 01-995792 has an invalid date: 04/31/1967
The running balance of all discarded records is 965.66
Account Number 02-168479 has an invalid date: 06/31/1956
The running balance of all discarded records is 984.25
*** Execution End: [xmldb:ref://////
MapProps_GlobalVariables.tf.xml] (version 1.0) completed
successfully
```

Note that the running balance increases with each discarded record as that record's balance is added to the **varBalance** variable during processing.

Reference

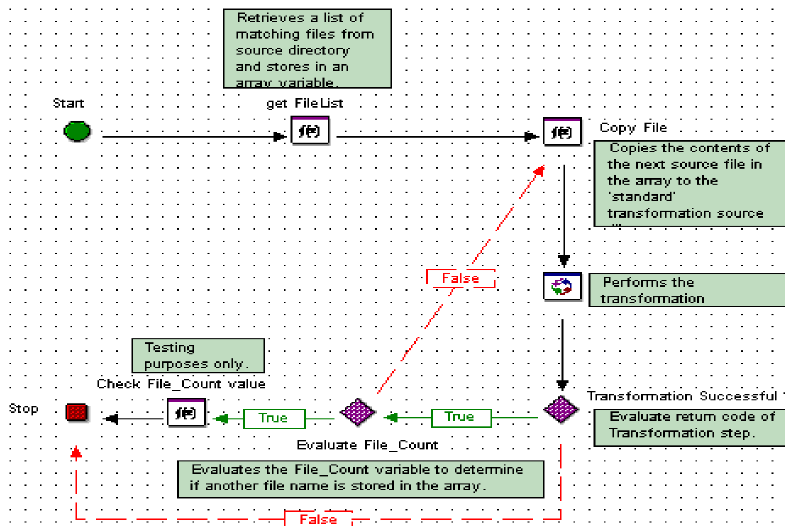
See “Variables” in the *RIFL Programmer’s Reference Manual*.

Using the FileList Function in a Process

This sample demonstrates how to check a directory for the existence of ASCII files and then process the files found.

Objectives

Check a folder for files to process, process those files, and then check the folder again until all files are processed.



Skill Level

Intermediate

Skill Set and Experience

- Process Designer
- Map Designer
- RIFL Scripting
- Basic understanding of arrays
- Basic understanding of how to order a process that includes decision steps

Design Considerations

To use Process Designer to perform **FileList** operations, you must plan the following in advance:

- *Where is the directory that holds the files to be processed located?*
  You need this information to connect to this directory. Important considerations are:
  - Does the directory reside on the same machine as the FileList process?
  - Does the directory reside on a machine other than the machine where the FileList process is executed?
  - If the directory resides on a machine other than the machine where the FileList process is executed, you must know the path to that machine and have permission to access it.
- *Where is the target of the transformation in the FileList process located?*
  Follow the considerations listed in the previous item.

Sample Process Location

SamplesDir`\Processes\FileList.ip.xml`

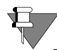
Sample Repository Configuration

The samples use workspaces and repositories to access the sample files. You must define that repository before running a sample transformation or process. For more information, see “Define a Samples Repository” in the “About the Samples” section.

Other Files Location

The FileList process uses the following transformation file:

SamplesDir`\Transformations\FileList.map.xml`



Note src_sourcefile.asc is only a temporary repository for data gathered from the files listed in the array created by the **getFile** scripting step.

The target for the transformation in the FileList process is an Access 97 table. This table is appended by the transformation.

Procedure Explanation

To accomplish this process in Process Designer, the following was done:

1 **Use a getFileList Step**
In the FileList process, the first step checks a named directory for files that have a .asc file extension. When the **getFile** step finds .asc files, it builds an array of the file names.

This scripting step obtains a list of all files in a directory. The expression in this step uses asterisks as wildcards. We named the step **getFileList** and entered the following RIFL expression:

'Declare the FileList array

```
Public File_List()
```

'Declare the loop counter

```
Public File_Count
```

'Use FileList Function to retrieve list of files in a specified directory and store in the array

```
FileList("Samples\Data\*.asc", File_List)
```

'Initialize the counter variable at the lower bound value of the array

```
File_Count = LBound(File_List)
```

'Test results of FileList to verify source files exist.

```
If File_Count > UBound(File_List) Then

 LogMessage("Warning","No source files exist at time
of execution: " & Now())

  Abort()

Else

'Display FileList results:

MsgBox("LBound Value = " & LBound(File_List) &
Chr(13) & Chr(10) & "Filename of LBound index = " &
File_List(LBound(File_List)) & Chr(13) & Chr(10) &
"UBound Value = " & UBound(File_List) )

End If
```

2 **Copy File**
This step copies the data in each of the .asc files in the array created by the **get FileList** step into a temporary file named **src_sourcefile.asc**.

The generic, or *standard*, source filename used in the transformation remains unchanged. We included code in this step to increment the looping **File_Count** variable.

' Location of the source files.

```
Private FilePath
       FilePath = "Samples\Data\"
```

' Copies contents of 'real' source file to 'standard' source file, named "sourcefile.asc", so that this file name can be used repetitively in the transformation.

```
FileCopy(FilePath & File_List(File_Count), FilePath
& "sourcefile.asc")
```

' Increment the File_Count variable

```
File_Count = File_Count + 1
```

3 **Convert Files**
The next process step is a Transformation map that performs an append to an Access 97 table. This map connects to an Access 97 database and appends to a table in:

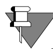
SamplesDir`\Data\sampleswork.mdb`

For this sample, the transformation map uses *straight mapping* where each source field is mapped directly to its corresponding target field without manipulation of data.

4 **Conversion Successful?**
This decision step generates a loop that checks if any files remain to be processed. If the **Convert Files** step was successful, we advance the process on the **True** branch. The code to evaluate the previous transformation success is:

```
project("Convert Files").ReturnCode = 0
```

If the **Convert Files** step was not successful, we advance the process on the **False** branch.



Note The **False** branch in this decision step advances the process to the **Stop** step.

5 **Evaluate File_Count**
In our sample the transformation was successful, so the process continues to this step. This decision step determines if all of the files in the **FileList** array in Step 1 of this procedure were processed.

This decision step checks to determine if the file count is greater than the upper bound of the **FileList** array using the following simple expression:

```
File_Count > UBound(File_List)
```

If this expression returns **false**, the process loops back to the **Copy File** step for further processing.

If this expression returns **true**, the process advances to the next step in sequence.

6 **Check File_Count value**
With a valid **File_Count** value, we generate a message box using the following expression:

```
MsgBox("File_Count = " & File_Count & chr(13)&chr(10)
& "UBound File_List = " & UBound(File_List))
```

This message displays the number of files remaining to be processed.

Note In the sample process, this expression is commented out because it requires user action to dismiss it. This step is used for testing purposes only.

7 **Stop Process**
The process advances to the **Stop** step and processing terminates.

More Detailed Information

The FileList process is a good example of a potential automated process. You can set a scheduler to start the process at intervals that meet your business process requirements.

Mapping Database Records to EDI

10

Electronic Data Interchange (EDI) uses standard formats to pass consistent data between disparate business systems. In this sample, we map records from a Microsoft Access database to an EDI target file.

Objectives

Use two tools to define and map the records from a Microsoft Access 97 database to predefined EDI target tables and fields. First use Document Schema Designer to select the EDI segments (record types) from a library. Then, using Map Designer, define the target fields, set up event handlers, and run the map.

Skill Level

Intermediate

Skill Set and Experience

- Map Designer
- Document Schema Designer
- Basic RIFL Scripting

Sample Map Location

SamplesDir`\Transformations\EDI_Mapping_SQL_to_EDI.map.xml`

Sample Repository Configuration

The samples use workspaces and repositories to access the sample files. You must define that repository before running a sample transformation or process. For more information, see "Define a Samples Repository" in the "About the Samples" section.

Source Information

Connection
Source Connector: **Access 97**
Source File/URI:
SamplesDir`\Data\Requester_270.mdb`
Source Table: **tblPatient**

Description
In this Access database we obtained the patient's first, middle and last name from the **tblPatient** table.

Target Information

Connection
Target Connector: **EDI (X12)**
Target Schema: **ControlSegments.ds.xml**
Output Mode: **Replace File/Table**
Target File/URI:
SamplesDir`\Data\ControlSegments.edi`

Description
The target EDI file contains seven record types. Only the **NM1** record type contains patient information after the map runs. The remaining record types are used for descriptive information about the interchange, group, and transaction.

Procedure Explanation

This sample uses two tools to transform the database content to EDI format — Document Schema Designer and Map Designer.

Part 1. Document Schema Creation

1. We first created a Document Schema to identify the EDI library segments that were appropriate for this data. After opening Document Schema Designer, we started a new document, indicated that no template would be used, and we selected **X12** as the schema type.
2. We used **Select Segment Library** to open the segment library containing the X12 library segments for EDI formatting.
3. We used the **Import Segment** option to import the following standard EDI library segments:

   ISA — Interchange Control Header
   GS — Functional Group Header
   ST — Transaction Set Header
   NM1 — Individual or Organizational Name
   SE — Transaction Set Trailer
   GE — Functional Group Trailer
   IEA — Interchange Control Trailer

   Most of these segments are used to set up the file start and end processing. Only the **NM1** segment is used to process individual or organization records.
4. We saved the new document schema as **ControlSegments** and used the **Create Sample Data File** option to create a target data file with the same name.

Part 2. Transformation Configuration and Processing

1. After setting up the connections in Map Designer, we used drag-and-drop functionality to copy the following records from the R1 source table to the **Target Field Expression** cells of the **NM1** target table on the **Map** tab:

Source Fields	Target Fields
subNameFirst	NM1_04_1036
subNameMiddle	NM1_05_1037
subLastOrgName	NM1_03_1035

2. In the source section, we added three **AfterFirstRecord** event handler **ClearMapPut Record** action to write a single instance of the **ISA**, **GS**, and **ST** header information at the beginning of the transformation.
3. To process the individual patient records, we used the source **AfterEveryRecord** event handler with **ClearMapPut Record** on the R1 source.
4. For the target **ST** record type, we added the **AfterPutRecord** event handler in the target section. This action resets the counter to 1 to account for the current **ST** segment after starting the new transaction.
5. Finally, we defined the remaining **Target Field Expression** cells for each record in each target record type.

After running the transformation, you can see that the header records (**ISA**, **GS**, and **ST**) appear only once at the beginning of the target file. The **NM1** record type repeats for each individual or organization record in the database. And the trailer records (**SE**, **GE**, and **IEA**) appear only once at the end.

More Detailed Information

While identifying and defining the appropriate EDI library segments may seem a little daunting in this brief sample description, it is less intimidating if you are accustomed to working with the EDI format and specifications. For more information on the EDI (X12) standards and structure, see the Accredited Standards Committee X12 website at http://www.x12.org/

Reference See “EDI (X12)” in the *Source and Target Connectors User’s Guide* available with your product.

Setting OnDataChange Events

11

Files that are sorted by one or more data items can also be monitored for a change of data values to identify a new grouping. In this sample we group records together and perform actions on those related records.

Objectives

Monitor a list of account entries by state and add the individual account balances together to generate a subtotal for each state.

Skill Level

Intermediate

Skill Set and Experience

- Map Designer
- RIFL Scripting

Design Considerations

The method used for **OnDataChange** event processing depends on the source format and sequence. If the source is already sequenced in the appropriate order for each group, no additional sorting is required (see "Part 1. Sorted Text File to Excel 97 Spreadsheet"). However, if the source is in random order, additional sorting must take place (see "Part 2. Unsorted Text File to Excel 97 Spreadsheet").

Sample Map Location

SamplesDir`\Transformations\Events_Part1_OnDataChange_map.xml`

SamplesDir`\Transformations\Events_Part2_OnDataChange_map.xml`

Sample Repository Configuration

The samples use workspaces and repositories to access the sample files. You must define that repository before running a sample transformation or process. For more information, see "Define a Samples Repository" in the "About the Samples" section.

Source Information

Connection
Source Connector: **ASCII (Delimited)**
Source File/URI:
Part 1: Sorted Source
SamplesDir`\Data\Accounts_SortedbyState.txt`

Part 2: Unsorted Source:
SamplesDir`\Data\Account.txt`

Description
The unsorted source file (**Accounts.txt**) contains the account records in random order. These records must be sorted into state sequence in order to perform the calculations needed for this sample. The sorted source file (**Accounts_SortedbyState.txt**) is already sorted by state name and does not require additional sorting to use the **OnDataChange** event handler.

Target Information

Connection
Target Connector: **Excel 97**
Output Mode: **Replace File/Table**
Target File/URI:
SamplesDir`\Data\AccountSummarybyState.xls`

Description
The target file is an Excel spreadsheet that is overwritten each time the transformation runs. This spreadsheet contains only three fields: **State**, **Number of Accounts**, and **Total Balance of Accounts**. The second and third fields do not appear in either source file and are combinations of record values from other source fields.

Procedure Explanation

In this sample we look at the steps for compiling source field values to generate new target field values for each state. The target field values include:

- **State** lists the abbreviation for the accounts from that state. This value is the same value as the source file **State** value.
- **Number of Accounts** provides a total number of accounts for the state listed in the target record. This value is generated by counting accounts in each state during the transformation.
- **Total Balance of Accounts** provides a total for the account balances for records in that state. This value is generated by adding together the **Balance** field value for the records in each state.

The following sample explanations vary slightly because the source files are different. Part 1 uses a sorted text file and Part 2 uses an unsorted text file. When processed correctly, both should provide the same results in the target spreadsheet.

Part 1. Sorted Text File to Excel 97 Spreadsheet

Map Name: **Events_Part1_OnDataChange.map.xml**

1 We connected the source and target files. Note that the source file (**Accounts_SortedbyState.txt**) is in ascending order by state name. Because the source records are already in the desired sequence, we do not need to perform any sorting for this transformation.

Masked Text 500

State	Zip	Email	Birth Date	Favorites	Standard I	Payments	Balance
AK	99775-6480	ffbla@uaf.com	07/02/1945	GA	112.00	120.85	120.85
AK	99901-0001	robert@alaska.net	03/22/1942	CA	121.00	121.00	212.12
AL	35294-1170	areeg@operamail.com	10/30/1967	KA	117.00	117.00	170.86
AR	71601	wilder@msn.com	07/03/1957	SB|FA	162.00	400.00	627.01
AZ	85224-4859	swilliams@nathanelec.com	12/19/1969	BA	150.00	0.00	0.00
AZ	85224-3987	scott@cgcmail.net	10/17/1959	RA	139.00	200.00	398.79
AZ	85013-2332	tom456@pcmail.net	12/01/1967	CA|RA|CC	124.00	125.00	243.25
AZ	86011-5717	alm53@ana.tx	06/31/1956	GA	101.00	18.59	18.59
AZ	85044-0098	carol@mcmail.net	02/14/1954	MA	113.00	113.00	136.34
AZ	85709-4453	armenta@pima.net	04/15/1954	SB	183.00	183.00	834.43
CA	91106-	aragon@paccd.net	08/12/1951	RA|MA	103.00	103.00	39.72
CA	92407-4657	mmonroe@mountain.com	10/13/1943	CA|RB	120.00	0.00	0.00
CA	92407-203	hmudd@startrek.com	04/26/1943	TA|MA	126.00	126.00	268.73
CA	91125-4958	sever@cco.net	04/24/1947	FA|KA	119.00	119.00	193.74

2 We used the **AfterEveryRecord** event handler to define the **Execute** action and set up variables for collecting and computing record values. This action uses the following expressions (explained in comment lines above each line):

' Set the state value for the current record because it will be different "OnDataChange"

```
varState = Records("R1").Fields("State")
```

' Increment the counter for the number or records within this block

```
varCounter = varCounter + 1
```

' Accumulate the balance for the records within this block

```
varBalance = varBalance +
Records("R1").Fields("Balance")
```

3 We used **Data Change Events** to set up the **OnDataChange1** event with two actions:

- **ClearMapPut Record** provides the standard source and target data processing.
- **Execute** resets the variable counters whenever a new state name is encountered:

  ' Reset these vars for next block of records

  ```
  varCounter = 0
  varBalance = 0
  ```

4 We added three target fields to use the variables set in the source to receive the transformation values:

(Excel 97) Target Record Type: R1							
	Target Field Name	Target Field Expression		Description	Type	Size	Results
1	State	=varState			Text	16	<null>
2	Number_of_Accounts	=varCounter			Text	16	<null>
3	Total_Balance_of_Accounts	=varBalance			Text	16	<null>
*							

5 We changed the source Balance field data type from Text to **Decimal**.

6 We validated the map and ran it. The following excerpt is a portion of the resulting target output:

Record No	State	Number_of_Accounts	Total_Balance_of_Accounts
1	AK	2	332.97
2	AL	1	170.86
3	AR	1	627.01
4	AZ	6	1631.4
5	CA	25	7636.12
6	CO	3	820.82
7	CT	2	754.84
8	DC	2	498.07
9	DE	1	57.07
10	FL	10	3269.59
11	GA	7	1568.49
12	HI	2	46.32
13	IA	2	1004.04
14	ID	1	760.5
15	IL	10	4933.69

Notice that all records for each state have been combined into a single record listing the state, the total number of accounts that were processed for that state, and the total of the balances for those accounts.

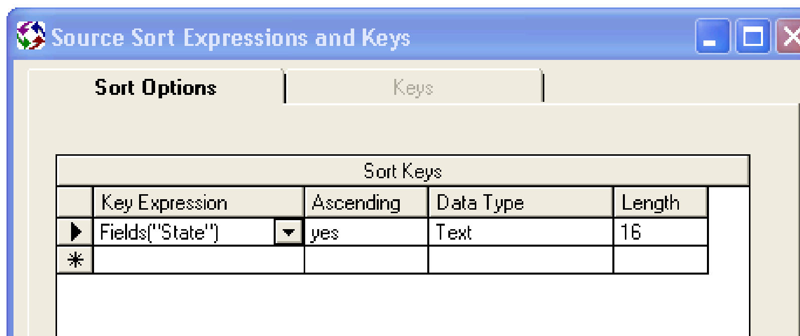
Part 2. Unsorted Text File to Excel 97 Spreadsheet

Map Name: **Events_Part2_OnDataChange.map.xml**

1. We connected the source and target files. Note that the source file (**Accounts.txt**) is in random order. If we processed the source without performing any sorting, the results would be incorrect because the states must be grouped together before **OnDataChange** can accurately identify the beginning of the next group and perform the required calculations.

State	Zip	Email	Birth Date	Favorites	Standard I	Payments	Balance
OH	44060-1930	warmst864@aol.com	02/28/1971	XA	101.00	100.00	15.89
IL	60637-4596	mapper@tcent.net	06/30/1975	BA	144.00	144.00	449.92
OH	44325-4002	dram@akron.net	11/02/1941	EB|CB	126.00	126.00	262.98
MA	2453	hazel@bentley.net	12/15/1962	JA|RB	127.00	127.00	271.75
LA	70803-4918	din33@norl.com	12/22/1985	ED|EA|RB|KA	142.00	150.00	423.01
SC	29303-9398	cadair@gw.com	02/21/1940	SB	151.00	155.00	515.41
GA	30535-1177	delores@truett.com	01/31/1960	MA|ED|SB	113.00	120.00	131.89
NY	12561-0023	dams@matrix.net	06/19/1940	EC	147.00	147.00	477.09
MD	20850-2080		02/23/1978	FA	103.00	100.00	32.31
CA	91106-	aragon@paccd.net	08/12/1951	RA|MA	103.00	103.00	39.72
WI	54702-4800	mary2@aol.com	02/22/1982	RB	103.00	103.00	34.21
IN	47374-9345	mary2@sumner.org	02/29/1974	CB	123.00	119.19	239.18
CO	81506-3493	classicwood@earthlink.net	12/07/1969	AC	107.00	99.98	75.09
VA	22031-3445	sjohnson@earthlink.net	11/07/1969	AC	107.00	0.00	0.00
IL	61821	ordin@parkland.net	12/15/1968	SA	171.00	175.00	710.81
FL	32608-3349	manone@hotmail.com	09/26/1971	CC	126.00	100.00	267.74
IL	60637-8869	chu@finmath.com	05/05/1954	LA	160.00	400.00	607.09
TX	78213-3318	fbkenhu@hotmail.com	09/05/1978	XA|TA	127.00	100.87	277.06
NC	28216-2876	jadeyeye@jcsu.net	10/04/1968	ED	140.00	150.00	408.43

2. We used the **Source Keys and Sorting** function to include sorting in the transformation:



This sorting takes place after the source is read, but before the data is processed by the **OnDataChange** event. This allows the event to properly identify the end of each group and trigger the necessary calculation steps.

3. After adding the sort options, we performed the same steps from steps 2 through 6 in “Part 1. Sorted Text File to Excel 97 Spreadsheet” to complete the transformation. The results of this transformation should be the same as the results of the Part 1 sample.

Reference

See “Event Handling” in the *Intermediate and Advanced Mapping User’s Guide.*

Using Buffered Put Tree to Create Hierarchical Records 12

Learn how to buffer a set of records and write the set to a file based on the hierarchical structure.

Objectives

Recursively walk through a hierarchical Source tree and write a set of records and their structure to an XML target file that supports a hierarchical layout. In this sample, the target will contain records grouped first by account number, then by parent records (customer information), and finally by dependent records.

Skill Level

Intermediate

Skill Set and Experience

- Map Designer
- XML Basics

Design Considerations

When using a **Put Tree** action for a hierarchical record layout (such as XML), you only need to specify the parent in the hierarchy to cause the children to be written in the appropriate order.

When you use the **Put Tree** action for a multirecord file with an implied hierarchical relationship, such as a fixed ASCII file, you must plan your map carefully. Map Designer does not define the parent-child relationship, so it cannot interpret the relationships between the parent and child tree buffers to write them out in a specific order. You must provide the logic in the map configuration.

Sample Map Location

SamplesDir`\Transformations\BufferedPutTree.map.xml`

Sample Repository Configuration

The samples use workspaces and repositories to access the sample files. You must define that repository before running a sample transformation or process. For more information, see “Define a Samples Repository” in the “About the Samples” section.

Source Information

Connection
Source Connector: **ASCII (Delimited)**
Source File/URI:
SamplesDir`\Data\src_BufferedPut.asc`

Description
The source ASCII file contains 83 records of account information.

Target Information

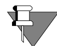
Connection
Target Connector: **XML**
Output Mode: **Replace File/Table**
Target File/URI:
SamplesDir`\Data\trg_BufferedPut.xml`

Description
Before the map runs, the target XML file contains fields that are placeholders to store multiple records and structure of the source file. When the transformation is successful, records are written to the appropriate fields and grouped in a hierarchical format when the <**All**> view is selected.

Procedure Explanation

To accomplish this transformation in Map Designer, the following was done:

1 We set up our source and target connections as described above.



Note It is not necessary to sort the source records because they are already in account number sequence. If your source is not already sequenced in the order in which you are building the hierarchical output, you must set up source key sorting before running the map.

2 On the **Map** tab, in the target grid, we set up three record types: **AccountInfo**, **CustomerInfo**, and **DependentInfo**. These record types are set up in the same hierarchy as the source records.

- **AccountInfo** is the top level and contains the **AccountNo** field. It also contains a nested secondary record type of **CustomerInfo**.
- **CustomerInfo** contains the **ParentFirstName** and **ParentLastName** fields to identify the customer group. It contains another nested record type of **DependentInfo**.

- **DependentInfo** is the lowest level record type and contains the most information. This record type contains the name and address fields for all “dependent” records.

3 Back on the source grid at the top, we set up two general event handlers to perform the recursive reading and writing steps:

   a. To process the first record, we added the **AfterFirstRecord** event with a **ClearMapPut Record** and set the **Buffered** flag to **False**. This action clears and writes the first record to the target.

   b. To process the remaining records in the source, we added an **AfterEveryRecord** event with **ClearMap** and **ClearMapPut Record** actions to clear and write **CustomerInfo** and **DependentInfo** to the target. We set the **Buffered** flag to **True** in the **ClearMapPut Record** action as well.

4 We set up the Source **OnDataChange1** event to define the actions to take place when the **Data Change Monitor** value changes (top of **Map** tab). We set that value as **Fields("Account No")** because the account number identifies the end of one group of records and the beginning of the next group. When the monitored data (account number) changes, the following actions take place:

   a. **PutRecord** writes the **CustomerInfo** to the target.

   b. **PutTree** writes the **DependentInfo** associated with the parent **CustomerInfo** record to the target. This writing continues until the monitored data changes again.

   c. **ClearTree** clears the current tree, clearing both fields and the buffer.

   d. **ClearMapPut Record** writes the **AccountInfo** to the target (remember, the **AfterFirstRecord** event writes only the first **AccountInfo** from the first record to the target).

5 Each time the account number changes in the source records, processing loops through the event handlers and continues to the end of the source records.

More Detailed Information

A successful transformation results in the fully-loaded XML target file with three record types containing the appropriate records and fields for each type:

- **AccountInfo** contains only the 17 different account numbers:

Record No	AccountNumber	CustomerInfo
1	10019	<null>
8	10028	<null>
15	10035	<null>
22	10041	<null>
29	10047	<null>
36	10054	<null>
43	10061	<null>
50	10067	<null>
57	10073	<null>
64	10080	<null>
71	10086	<null>
78	10092	<null>
85	10099	<null>
92	10105	<null>
99	10111	<null>
106	10117	<null>
113	10123	<null>
118	<null>	<null>

- **CustomerInfo** also has 17 records listed because, in this case, each customer has a single account number and visa versa.

Record No	FirstName	LastName	DependentInfo
2	Bobbi	Arndt	<null>
9	Carol	Braun	<null>
16	Connie	Catterton	<null>
23	Darlene	Dellenmann	<null>
30	Dick	Dunbar	<null>
37	Duane	Feavel	<null>
44	Glenn	Gorman	<null>
51	Jane	Haugner	<null>
58	Jim	Issacson	<null>
65	John	Kamp	<null>
72	Joyce	Kretsch	<null>
79	Larry	Lesperance	<null>
86	Lori	Ludwig	<null>
93	Marty	Meyere	<null>
100	Mary	Novak	<null>
107	Mike	Phillips	<null>
114	Pam	Richeson	<null>
119	<null>	<null>	<null>

- **DependentInfo** has many more records because it lists all the records for the dependent fields and each customer happens to have 5 dependents.

Record No	FirstName	LastName	Address	City	State	Zip
3	Bruce	Beecher	1037 W Wisconsin Ave	Smithville	AK	99140-1599
4	Bruce	Beyer	108 E Wisconsin Ave	Jonestown	AK	99143-1803
5	Butch	Bobbi	108 Hillock Ct	Smithville	AK	99166-3208
6	Calla	Boshers	110 Fox River Dr	Smithville	AK	99166-3425
7	Carol	Brauer	110 W North Water St	Jonestown	AK	99216-1144
10	Cheri	Buksyk	1122 Milwaukee St	Jonestown	AK	99401-4175
11	Chuck	Buss	1134 S Franklin St	Overton	AK	99401-9903
12	Chuck	Carpenter	115 S Drew St	Smithville	AK	99433-9418
13	Chuck	Carr	1151 Valley Fair Mall	Jonestown	AK	99451
14	Colleen	Casperson	120 N Morrison St	Smithville	AK	99501
17	Connie	Clay	121 N Douglas St # R	Smithville	AK	99901
18	Craig	Collar	1216 W Wisconsin Ave	Moretown	AK	99901
19	Dan	Coppenger	1221 N Lawe St	Smithville	AK	99901-2104
20	Dan	Dag	1222 N Superior St	Smithville	AK	99901-2990
21	Darlene	Dantzier	124 W Wisconsin Ave	Dime Box	AK	99901-4848
24	Darren	Dinkl	1300 E Calumet St	Smithville	AK	99903-2099
25	David	Dockry	1302 S Ritger	Smithville	AK	99903-2692
26	Debra	Dorsey	1320 S Lincoln St	Smithville	AK	99903-3065
27	Denise	Richardson	1396 Ridgeway Court	Smithville	AK	99911
28	Diane	Duginski	144 N Mall Dr	Smithville	AK	99911
31	Dinah	Duxbury	1486 Earl St	Dime Box	AK	99911
32	Dominic	Earl	150 W Green Bay Rd	Lonesome	AK	99911
33	Don	Ely	1620 S Lawe St	Smithville	AK	99911
34	Donna	Erdmann	1737 W Reid Dr	Smithville	AK	99911
35	Donna	Esser	1801 N Richmond St #	Jonestown	AK	99911
38	Emmett	Forseth	1818 N Meade St	Smithville	AK	99911-2781

You can see the actual hierarchy of account numbers, customer information, and dependent information in the <**All**> view of the target data:

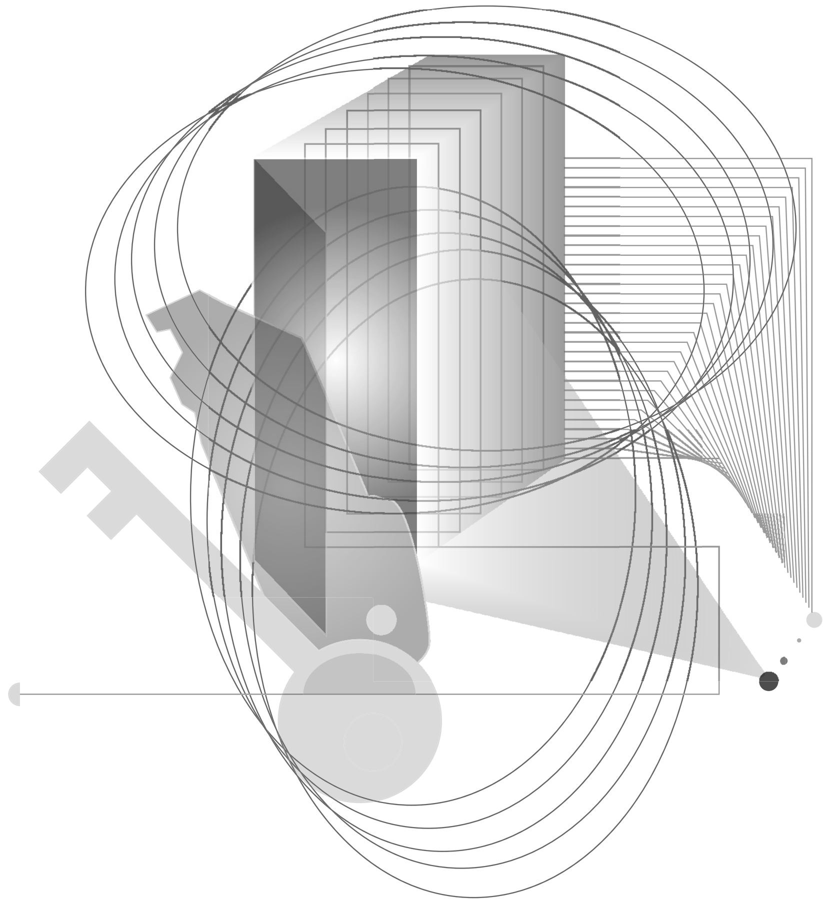
Record No	Record Name	Field Name	Field Contents
1	AccountInfo	AccountNumber	10019
		CustomerInfo	<null>
2	CustomerInfo	FirstName	Bobbi
		LastName	Arndt
		DependentInfo	<null>
3	DependentInfo	FirstName	Bruce
		LastName	Beecher
		Address	1037 W Wisconsin Ave
		City	Smithville
		State	AK
		Zip	99140-1599
4	DependentInfo	FirstName	Bruce
		LastName	Beyer
		Address	108 E Wisconsin Ave
		City	Jonestown
		State	AK
		[illegible]	[illegible]

On this page you see that **AccountInfo** and **CustomerInfo** repeats each time the account number changes. The **DependentInfo** records below each **CustomerInfo** record are the children of that record. This type of hierarchical group is repeated for each set of account number and customer information records.

Reference

See “Event Actions” in the *Intermediate and Advanced Mapping User’s Guide.*

For an example of how to set up hierarchical target record types and fields, see “Map Designer Tutorial 5 - Single-Record Type File to a Multirecord Type XML File” in the *Tutorials Reference.*



ADVANCED SAMPLES

Aggregating Records

13

Using Map Designer, this sample demonstrates how to aggregate values from multiple records of a single record type.

Objectives

Aggregate account information from multiple customer records of a single record type.

Skill Level

Advanced

Skill Set and Experience

- Map Designer
- RIFL Scripting
- Event Handlers
- Event order of precedence

Design Considerations

The source file contains customer purchase records of a single record type. The goal is to aggregate that customer information. This requires that we consider which of the customer account fields we want to combine. For our sample, we aggregated customer purchases into a customer purchase history format.

Another important prerequisite after design is to have a thorough knowledge of event handlers and especially event precedence.

Sample Map Location

SamplesDir`\Transformations\Record Aggregation.map.xml`

Sample Repository Configuration

The samples use workspaces and repositories to access the sample files. You must define that repository before running a sample transformation or process. For more information, see "Define a Samples Repository" in the "About the Samples" section.

Source Information

Connection
Source Connector: ASCII (Delimited)

Source File/URI:
SamplesDir`\Data\src_tutor1.asc`

Description
The source file contains 100 records with fields that contain customer contact information and account balances.

Target Information

Connection
Target Connector: **ASCII (Fixed)**
Output Mode: **Replace File/Table**
Target File/URI:
SamplesDir`\Data\trg_aggregate.asc`

Description
The target file contains source file values aggregated into totals and averages by city and state.

Procedure Explanation

Although the concept behind this transformation is easy to comprehend, the skills required to construct it are advanced. This transformation consists of some simple nested **If** expressions that are fired in a specific order to populate the target. It is this firing order that adds complexity. Within the transformation, we read the source file and aggregate totals and averages by city and state.

Line Breaks in Sample Code

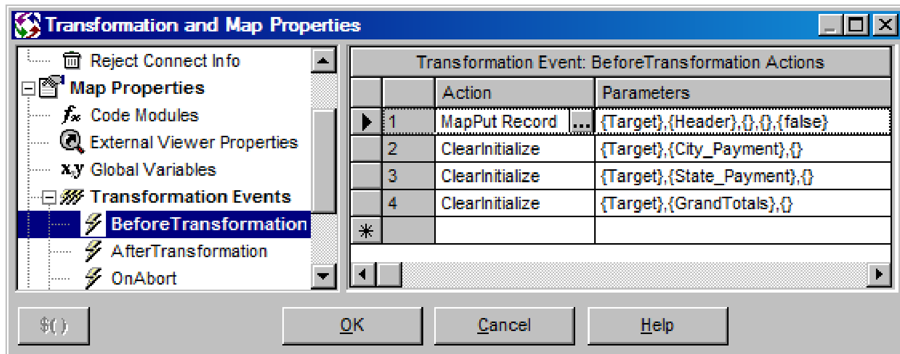
In this sample procedure you will find the RIFL script expressions we used to calculate and populate target fields. Due to space available, some of the expressions wrap in this documentation in a manner that renders the expression invalid in the RIFL Script Editor. The RIFL Script Editor is the ultimate authority of whether or not a RIFL expression is valid. Bear these rules in mind:

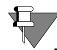
- Lines that start with **If** must close with **Then**.
- Lines that start with **ElseIf** must close with **Then**.
- **Else** must be on a line by itself.
- **End If** must be on a line by itself.

If you are not sure of the correct breaking point for the lines of code shown in this sample, please look at the actual code in Map Designer to see how it is handled there.

Performing the Transformation

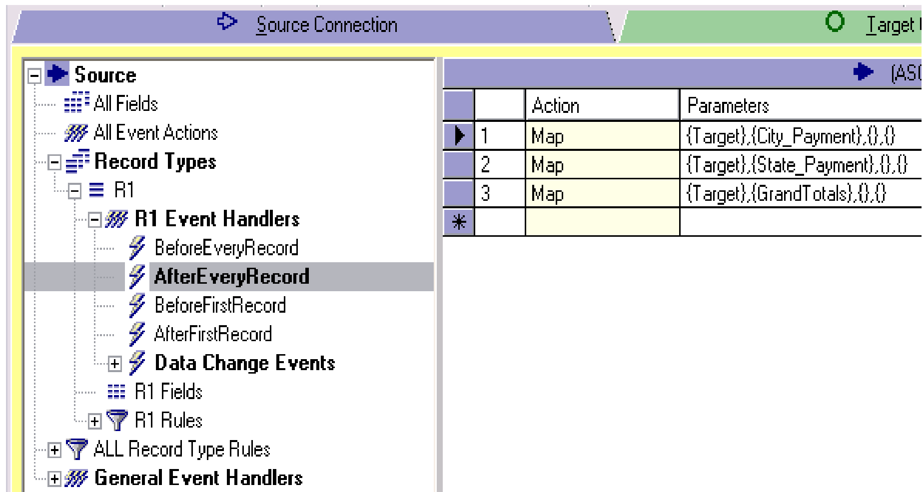
1. To begin aggregating records from the source file, we opened Map Designer, selected **New Map** from the **File** dropdown menu, and connected to the source file.
2. In **Transformation and Map Properties**, we set four **BeforeTransformation** actions — one **MapPutRecord** and three **ClearInitialize**. See the following graphic for the properties we set.





Note The **ClearInitialize** action clears the buffer of target records and initializes the non-null field values. The **ClearInitialize** action also initializes all numeric and text fields to zero. This prevents null field values from skewing the target values. For example, **null** + **2** = **null**, but **0** + **2** = **2**.

3. On the **Map** tab, we set a source **AfterEveryRecord** event handler to map the output to the target. See the following graphic for detail on the **AfterEveryRecord** event handler configuration.



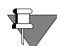
4 Next, on the **Map** tab, we set two source **OnDataChange** event handlers to control when records are written to target. The first **OnDataChange** calculates the value of **C_Pymnt_Avg** using the following expression:

```
dim avg
avg =
Targets(0).Records("City_Payment").Fields("C_Pymnt_Total")
/
Targets(0).Records("City_Payment").Fields("C_Pymnt_Count")
Targets(0).Records("City_Payment").Fields("C_Pymnt_Avg") =
format(avg, "#.00")
```

We set this first **OnDataChange** event handler to suppress the first firing to prevent writing a null value, and fire an extra event at end of file to clear the final value from the buffer and write it to target.

5 We set two additional event actions for the first **OnDataChange** event.

- **Put Record** is set for a non-buffered **Put** of **City_Payment**.
- **ClearInitialize** is set for the target field **City_Payment**.



Note The **ClearInitialize** action clears the buffer of target records and initializes the non-null field values.

6 The second source **OnDataChange** event handler calculates the value of **St_Pymnt_Avg** using the following expression:

```
dim avg
avg =
Targets(0).Records("State_Payment").Fields("St_Pymnt_Total
") /
Targets(0).Records("State_Payment").Fields("St_Pymnt_Count
")
Targets(0).Records("State_Payment").Fields("St_Pymnt_Avg")
= format(avg,"#.00")
```

We set this second **OnDataChange** to suppress the first firing to prevent writing a null value, and fire an extra event at end of file to clear the final value from the buffer and write it to target.

7. We set two additional event actions for the second source **OnDataChange** event:

   - **Put Record** is set for a non-buffered **Put** of **City_Payment**.
   - **ClearInitialize** is set for the target field **State_Payment**.

8. On the **Map** tab, in the Source tree **General Event Handlers**, we set an **OnEOF Execute** and non-buffered **Put Record** action

   - **Execute** uses an expression to calculate **ItemAvg**:

     ```
     dim avg
     avg =
     Targets(0).Records("GrandTotals").Fields("ItemTotal")
     /
     Targets(0).Records("GrandTotals").Fields("ItemCount")
     Targets(0).Records("GrandTotals").Fields("ItemAvg") =
     format(avg,"#.00")
     ```

   - **Put Record** was set for a non-buffered **Put** of target **GrandTotals**.

9. Still on the Target tree, we named and defined six **City_Payment** target fields as follows:

   - **CityName**

     ```
     Records("R1").Fields("City")
     ```

   - **C_Pymnt_Count**

     ```
     Val(Targets(0).Records("City_Payment").Fields("C_Pymnt
     _Count")+1)
     ```

   - **C_Pymnt_Total**

     ```
     Fields("Payment") +
     Targets(0).Records("City_Payment").Fields("C_Pymnt_Tot
     al")
     ```

   - **C_Pymnt_Min**

```
If
Targets(0).Records("City_Payment").Fields("C_Pymnt_Min
") = 0 Then
Fields("Payment")
ElseIf
Fields("Payment")<Targets(0).Records("City_Payment").F
ields("C_Pymnt_Min")
Then
Fields("Payment")
Else
Targets(0).Records("City_Payment").Fields("C_Pymnt_Min
")
End If
```

- **C_Pymnt_Max**

```
If
Targets(0).Records("City_Payment").Fields("C_Pymnt_Max
") = 0 Then
Fields("Payment")
ElseIf Fields("Payment")  >
Targets(0).Records("City_Payment").Fields("C_Pymnt_Max
") Then
Fields("Payment")
Else
Targets(0).Records("City_Payment").Fields("C_Pymnt_Max
")
End If
```

- **C_Pymnt_Avg** is populated by the first source **OnDataChange** event action.

10 We named and defined six **State_Payment** target fields as follows:

- **StateName**

```
"Totals for: " & Fields("State")
```

- **St_Pymnt_Count**

```
Targets(0).Records("State_Payment").Fields("St_Pymnt_C
ount") + 1
```

- **St_Pymnt_Total**

```
Fields("Payment")  +
Targets(0).Records("State_Payment").Fields("St_Pymnt_T
otal")
```

- **St_Pymnt_Min**

```
If
Targets(0).Records("State_Payment").Fields("St_Pymnt_M
in") = 0 Then
Fields("Payment")
ElseIf Fields("Payment") >
Targets(0).Records("State_Payment").Fields("St_Pymnt_M
```

```
in") Then
Targets(0).Records("State_Payment").Fields("St_Pymnt_M
in")
Else
Fields("Payment")
End If
```

- **St_Pymnt_Max**

```
If
Targets(0).Records("State_Payment").Fields("St_Pymnt_M
ax") = 0 Then
Fields("Payment")
ElseIf Fields("Payment") >
Targets(0).Records("State_Payment").Fields("St_Pymnt_M
ax") Then
Fields("Payment")
Else
Targets(0).Records("State_Payment").Fields("St_Pymnt_M
ax")
End If
```

- **St_Pymnt_Avg** is populated by the second source **OnDataChange** event action.

11 We named and defined six **GrandTotals** target fields as follows:

- **AllItems**

```
Targets(0).Records(4).Fields(1).Name
```

- **ItemCount**

```
Targets(0).Records("GrandTotals").Fields("ItemCount")
+ 1
```

- **ItemTotal**

```
If 0 =
Targets(0).Records("GrandTotals").Fields("ItemTotal")
Then
Fields("Payment")
Else
Targets(0).Records("GrandTotals").Fields("ItemTotal")
+ Fields("Payment")
End If
```

- **ItemMin**

```
If 0
=Targets(0).Records("GrandTotals").Fields("ItemMin")
Then
Fields("Payment")
ElseIf
Targets(0).Records("GrandTotals").Fields("ItemMin") <
Fields("Payment") Then
```

```
Targets(0).Records("GrandTotals").Fields("ItemMin")
Else
Fields("Payment")
End If
```

- **ItemMax**

```
If 0
=Targets(0).Records("GrandTotals").Fields("ItemMax")
Then
Fields("Payment")
ElseIf
Targets(0).Records("GrandTotals").Fields("ItemMax") >
Fields("Payment") Then
Targets(0).Records("GrandTotals").Fields("ItemMax")
Else
Fields("Payment")
End If
```

- **ItemAvg** is populated by the source **OnEOF** event handler.

Reference

See “Event Handling” in the *Intermediate and Advanced Mapping User’s Guide.*

Manipulating Binary Dates at the Bit Level

14

The most common data manipulation occurs at the byte level. Map Designer also allows you to manipulate data at the bit level through the use of the RIFL expression language. In this sample we convert a date at the bit level from a binary source file to a target database.

Objectives

Convert a date stored as a 16-bit (2 byte) binary source field to a valid date field in the target file using bit-level manipulation.

Skill Level

Advanced

Skill Set and Experience

- Map Designer
- RIFL Scripting
- Basic understanding of binary file structure

Design Considerations

To use Map Designer bit-level manipulation effectively, you must analyze some details in advance:

- *How is the source data stored?*
  You need this information in order to use the techniques described in this sample transformation.
- *Which bit of each byte holds the data you want to manipulate?*
- *Is the data stored as binary, hexadecimal, octal, or decimal?*
  This sample assumes the data is stored in binary format.

You must also have some concept of how the data is stored in bits and bytes. See “More Detailed Information” on page 14-5 for a description of bit and byte usage.

Sample Map Location

SamplesDir`\Transformations\Bit Level Manipulation of Dates.map.xml`

Sample Repository Configuration

The samples use workspaces and repositories to access the sample files. You must define that repository before running a sample

transformation or process. For more information, see “Define a Samples Repository” in the “About the Samples” section.

Source Information

Connection
Source Connector: **Binary**
Source File/URI:
SamplesDir`\Data\src_bit_manip.txt`

Description
The binary data file uses a 16-bit (2-byte) integer to represent a date. The source field was defined in Data Parser as a 2-byte binary field, so it is displayed in Source Data Browser in unpacked format (ASCII 5000) (see sample screen below).

For this example, we used only the **Field1** field in the source data file.

Record No	Field1
1	5000
2	5050
3	5100
4	5150
5	5200
6	5250
7	5300
8	5350
9	5400
10	5450
11	5500
12	5550
13	5600
14	5650
15	5700
16	5750
17	5800
18	5850
19	5900
20	5950
21	6000

For more information on working with binary files, see “More Detailed Information” on page 14-5.

Target Information

Connection
Target Connector: **dBASE IV**
Output Mode: **Replace File/Table**
Target File/URI:
SamplesDir`\Data\trg_bit_manip.dbf`

Description

The target is a dBase file containing a field that is a valid data type for a date. Thes dBASE file was set up to display the converted data in several ways, as described in the following table.

Table 14-1 Target dBASE File Layout

Column #	Column Name	Description
Column 1	SOURCE	Displays the source data converted from its unpacked format to a 4-byte character field “as is.” This was done merely to prove the accuracy of the date conversion.
Column 2	DATE	Displays the source data converted from its original binary format to a 4-byte **Date** field. This is the normal conversion you perform in most circumstances. Notice the data is stored in dBASE in a **yyyymmdd** format. This is the standard method of storing dates in dBASE. It is important to note the blank fields in this column. These are records where the source data is not a valid date. Note that in the month column the converted values do not fall between 01 and 12.
Column 3	YEAR	Here we converted the “year bits” in the source to a single target character field for comparison to the data in column 2.
Column 4	MON	Here we converted the “month bits” in the source to a single target character field for comparison to the data in column 2.
Column 5	DAY	Here we converted the “day bits” in the source to a single target character field for comparison to the data in column 2.

Procedure Explanation

We used the following steps to accomplish this transformation in Map Designer:

1. To avoid repeating the same functions (and their arguments) multiple times, global variables were declared in the **Transformation and Map Properties** as:
   - **Day1** *for* storing day data
   - **Month1** for storing month data
   - **Year1** for storing year data
2. To return the original value of the source as a character, the following expression was written in the **Target Field Expression** cell on the **Map** tab for the target field named **SOURCE**:

   ```
   Records("Record1").Fields("Field1")
   ```

3. To calculate the values of the day, month, and year in the record, the following expressions were written in the **Target Field Expression** cell on the **Map** tab for the target field named **DATE**:

   'This expression calculates the value of the day by comparing the bit values.

   ```
   Day1 = Records("Record1").Fields("Field1") And 31
   ```

   'This expression calculates the value of the month by comparing the bit values.

   ```
   Month1 = (Records("Record1").Fields("Field1")/32)
   And 15
   ```

   'These expressions calculate the value of the year by comparing the bit values and adjusting for single digit years.

   ```
   Year1 = (Records("Record1").Fields("Field1")/512)
   And 127

   If Len(Trim(Year1)) = 1 Then

     Year1 = 0 & Year1

   End If
   ```

4. To process the year as a four-digit character, the following expression was written in the **Target Field Expression** cell on the **Map** tab for the target field named **YEAR**:

   ```
   Year1 = (Records("Record1").Fields("Field1")/512)
   And 127

   If Len(Trim(Year1)) < 2 Then

     "190" & Year1
   ```

```
Else
  "19" & Year1
End If
```

5 To ensure the month is a two-digit character, the following expression was written in the **Target Field Expression** cell on the **Map** tab for the target field named **MON**:

```
Month1 = (Records("Record1").Fields("Field1")/32)
And 15
If Len(Trim(Month1)) < 2 Then
  "0" & Month1
Else
  Month1
End If
```

6 To return a four-digit character for use as the year, the following expression was written in the **Target Field Expression** cell on the **Map** tab for the target field named **DAY**:

```
Year1 = (Records("Record1").Fields("Field1")/512)
And 127
If Len(Trim(Year1)) < 2 Then
  "190" & Year1
Else
  "19" & Year1
End If
```

More Detailed Information

To work effectively with binary files, you must understand the concept of how data is stored in bits and bytes.

- A byte is made of eight (8) bits.
- Each of the 8 bits contains either a zero (0) or a one (1). Zero resolves as **false** and one resolves as **true**.
- The bits within a byte are read from right to left.

Example

The following is a representation of a 16-bit binary (2-byte) — hex = 0x13 0x88 — data field that is used to store a date. It illustrates the source data in record 1 (ASCII = 5000) in the example described at the beginning of this section.

```
0 0 0 1 0 0 1 1 1 0 0 0 1 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
64 32 16 8 4 2 1 8 4 2 1 16 8 4 2 1
Year Month Day
Day = 5000 And 31
Month = (5000 / 32) And 15
Year = (5000 / 512) And 127
```

This expression is an example of bit masking using the binary **And** operator.

The result is **12/08/1909.**

Complex Date Filtering 15

One of the more common reports you may want to produce is a report of records by date range. Map Designer allows you to extract records from a source and use a complex target filter to write only the records within a specified range of dates to a target file.

Objectives

Select a group of records filtered by a date range and write the records to a target file.

Skill Level

Advanced

Skill Set and Experience

- Map Designer
- RIFL Scripting
- Familiarity with date masks.

Design Considerations

To use Map Designer source filtering effectively, you must analyze some details in advance:

- *How is the date in each record formatted?*
  You need this information in order to use the techniques described in this sample transformation.
- *What date format do I want to use for the target file?*

Sample Map Location

SamplesDir`\Transformations\Complex Date Filtering.map.xml`

Sample Repository Configuration

The samples use workspaces and repositories to access the sample files. You must define that repository before running a sample transformation or process. For more information, see “Define a Samples Repository” in the “About the Samples” section.

Source Information

Connection
Source Connector: **ASCII (Delimited)**
Source File/URI:
SamplesDir`\Data\src_fwd_dates.asc`

Description

For this example, we used only the **Record Number** and **Date** fields. These happen to be the only two fields in the source file. If we had used a source file that contained more fields, the source filtering expressions would still be the same as those shown in this sample transformation, with adjustments for differing field names.

Record No	Recor	Date
341	341	12/07/1998
342	342	12/08/1998
343	343	12/09/1998
344	344	12/10/1998
345	345	12/11/1998
346	346	12/12/1998
347	347	12/13/1998
348	348	12/14/1998
349	349	12/15/1998
350	350	12/16/1998
351	351	12/17/1998
352	352	12/18/1998
353	353	12/19/1998
354	354	12/20/1998
355	355	12/21/1998
356	356	12/22/1998
357	357	12/23/1998
358	358	12/24/1998
359	359	12/25/1998
360	360	12/26/1998
361	361	12/27/1998
362	362	12/28/1998
363	363	12/29/1998
364	364	12/30/1998
365	365	12/31/1998

Target Information

Connection

Target Connector: **ASCII (Delimited)**
Output Mode: **Replace File/Table**
Target File/URI:
SamplesDir`\Data\trg_filtering1.txt`

Procedure Explanation

We used the following steps to accomplish this transformation in Map Designer:

1. After we connected to the source file, we clicked the **Source Filter** icon to open a dialog box for entering the RIFL expression that will perform the date filtering.

2. In the **Source Filtering and Samples** window we entered the following RIFL expressions to filter all dates between June and August:

```
DatePart("m", DateValMask(Fields("Date"),
"mm/dd/yyyy")) >= 6 And
DatePart("m", DateValMask(Fields("Date"),
"mm/dd/yyyy")) <= 8
```

3. The second expression in the window selects dates with valid numeric days (days from 1 to 31):

```
DatePart("d", DateValMask(Fields("Date"),
"mm/dd/yyyy")) >= 1 And
DatePart("d", DateValMask(Fields("Date"),
"mm/dd/yyyy")) <= 31
```

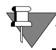
4. The final expression selects only the records with a year date of 1998:

```
DatePart("yyyy", DateValMask(Fields("Date"),
"mm/dd/yyyy")) = 1998
```

5. We closed the **Source Filtering and Samples** window, performed validation, and then ran the transformation.

More Detailed Information

The **DatePart** and **DateValMask** functions are used exclusively in this transformation to choose the portions of each record date and transform them to the desired format.



Note The mask used in **DateValMask (mm/dd/yyyy**) identifies the format of dates in the source file, not the desired target date format. All source field dates must be in this format for the function to return the appropriate data to the target.

Reference

See “DatePart Function” and “DateValMask Function” in the *Rapid Integration Flow Language (RIFL) Reference*.

Working with DJRowSet and Arrays

16

A **DJRowSet** object is a container for storing collections of record instances similar to an array. We use this sample to review the flexibility and ease of use of **DJRowSet** for handling “jagged arrays”.

Objectives

Use **DJRowSet** operations to segment text from a source file into three record types and insert the appropriate rows in the transformation target file.

Skill Level

Advanced

Skill Set and Experience

- Map Designer
- RIFL Scripting
- Basic concept of arrays for managing collections of data

Design Considerations

Buffered **Put and Put Tree** actions provide an alternate method of handling multirecord files with an implied hierarchical relationship. You may find, however, that **DJRowSet** is more appropriate when you need to write a parent node containing values that are not known until all relevant source records have been read. In this sample, we also use **DJRowSet** with the **OnDataChange** event handler to be activated when a new record is encountered.

Sample Map Location

SamplesDir`\Transformations\DJRowsetObject.map.xml`

Sample Repository Configuration

The samples use workspaces and repositories to access the sample files. You must define that repository before running a sample transformation or process. For more information, see “Define a Samples Repository” in the “About the Samples” section.

Source Information

Connection
Source Connector: **ASCII (Delimited)**

Source File/URI:
SamplesDir`\Data\Purchases_Phone_ASCII.txt`

Description
The source is a simple text file containing 36 records with all the pertinent information about a product purchase.

Target Information

Connection
Target Connector: **XML**
Output Mode: **Replace File/Table**
Target File/URI:
SamplesDir`\Data\Purchases_Hier.xml`

Description
The target file contains three data types, or tables, that were set up to contain specialized information about the product and purchase.

Procedure Explanation

We took data from a simple text file and transformed it into three tables (record types) in a target XML file.

1. We set up our source and target connectors as usual.
2. On the **Map** tab, in the target grid, we set up three new record types — **AccountInformation**, **PurchaseOrderInformation**, and **Item_Information**.

Target Records: (XML) -> $(Samples)Purchases_Hier.xml

	Record Name	Leng	Lock	Schema Origin	Description
1	AccountInformation	9	No		
2	PurchaseOrderInformation	54	No		
3	Item_Information	55	No		
*					

3. In the source grid, we set up an **AfterEveryRecord** event handler with a single **Execute** action. The **Execute** action does the following:

   a. Appends a row to the row set based on the current source record:

   ```
   myItems.Append(Sources(0).Records("R1").Self)
   ```

   b. Increments and aggregates the variables:

   ```
   varItemNumber = varItemNumber + 1
   varPOTotal = varPOTotal +
   Records("R1").Fields("Total")
   ```

Notice how these variable names correspond to the target record types set up earlier.

4 We set up the source **OnDataChange1** event handler to read and write the records from the source to the target:

   a. We added one **ClearMapPut Record** for **AccountInformation** and accepted the default settings.

   b. We added a second **ClearMapPut Record** for **PurchaseOrderInformation** and again accepted the default settings.

   c. For **Item_Information**, we also added the **ClearMapPut Record** action, but this time we added two other values:

      - For **count**, we specified **myItems.Size** to use the row option **Size** to capture the size of the entry.
      - For **counter variable**, we specified **cntr** to track the number of items.

Note The value in the **PONumber** field is used as the *data change monitor* in this sample. In the source file, all records for each purchase order are grouped together. When the value in that field changes as a new record is read, the **OnDataChange** event is triggered. See “Setting OnDataChange Events” for a sample of that functionality.

---

   d. The last action we added for this event handler was **Execute**. This action contains the following short script to truncate the row set table (**myItems**) and reset the variables to zero:

```
myItems.Clear
varItemNumber = 0
varPOTotal = 0
```

5 We copied the source fields to the target grid individually for each record type as follows:

- **AccountInformation**:
  AccountNumber
- **PurchaseOrderInformation**:
  PONumber
  PurchaseDate

- **Item_Information**:
  Category
  ProductNumber
  Quantity
  UnitCost
  Total
  ShipmentMethodCode

6 We added the following null fields to tie the tables together:

- **AccountInformation**:
  PurchaseOrderInformation
- **PurchaseOrderInformation**
  Item_Information

7 We added three more target fields to receive the variables defined in the source (steps 3 and 4c):

Target Field Name	Target Record	Target Field Expression
ItemNumber	Item_Information	=cntr
Number_of_Items	PurchaseOrderInformation	=varItemNumber
POTotal	PurchaseOrderInformation	=varPOTotal

8 We validated the transformation and ran it. The resulting target file contains three tables with the appropriate data:

- **AccountInformation** contains only the account numbers and a null field. Twelve account numbers are listed because there are twelve purchase orders associated with those accounts.

Record No	AccountNumber	PurchaseOrderInformation
1	02-385862	<null>
4	02-336632	<null>
8	02-336632	<null>
12	01-674085	<null>
15	01-674085	<null>
19	01-438824	<null>
23	01-438824	<null>
29	01-990288	<null>
32	02-109948	<null>
40	02-109948	<null>
46	02-792777	<null>
51	<null>	<null>

- **PurchaseOrderInformation** contains the information that describes the overall purchase order. Any duplicate purchase order numbers were removed. In this table the value of the **varPOTotal** variable is displayed. This variable kept a running tally of the total cost of each item in the purchase order and wrote that information in the **POTotal** column.

Record No	PONumber	PurchaseDate	Number_of_Items	POTotal	Item_Information
2	02-385862-01	12/28/2002	1	17.079999999999998	<null>
5	02-336632-01	12/17/2002	2	867.830000000000048	<null>
9	02-336632-02	12/18/2002	2	50.940000000000001	<null>
13	01-674085-01	12/23/2002	1	41.039999999999999	<null>
16	01-674085-02	12/24/2002	2	299.559999999999993	<null>
20	01-438824-01	12/17/2002	2	71.889999999999997	<null>
24	01-438824-02	12/18/2002	4	153.009999999999999	<null>
30	01-990288-01	12/08/2002	1	43.299999999999997	<null>
33	02-109948-01	12/10/2002	6	933.719999999999992	<null>
41	02-109948-02	12/11/2002	4	134.669999999999961	<null>
47	02-792777-01	12/20/2002	3	158.320000000000003	<null>
52	<null>	<null>	8	447.219999999999988	<null>

- **Item_Information** contains more records because it provides information about the individual items that were included in various purchase orders for different accounts. Notice that the **ItemNumber** column reflects the number that was assigned to each item as the record was processed. This count is the result of the **varItemNumber** variable.

Record No	ItemNumber	Category	ProductNumber	Quantity	UnitCost	Total	ShipmentMethodCode
3	1	CA	Y31-QE72	12	67.61	811.32	SUBOR
6	1	BA	M75006-WQSB4	1	34.74	34.74	LITBR
7	2	BA	M75006-WQSB4	1	34.74	34.74	LITBR
10	1	RB	RTR67-NV6	2	20.52	41.04	UPS2D
11	2	RB	RTR67-NV6	2	20.52	41.04	UPS2D
14	1	JA	D747933-TNYK25	3	19.84	59.52	USPSF
17	1	AC	I44159035-SAL78	1	50.30	50.30	FPOAP
18	2	AC	I44159035-SAL78	1	50.30	50.30	FPOAP
21	1	XA	BW4008-FQKE1	1	7.50	7.50	UPSS2
22	2	XA	BW4008-FQKE1	1	7.50	7.50	UPSS2
25	1	BA	I0201313060-CWF56	1	43.30	43.30	POUCH
26	2	BA	I0201313060-CWF56	1	43.30	43.30	POUCH
27	3	BA	I0201313060-CWF56	1	43.30	43.30	POUCH
28	4	BA	I0201313060-CWF56	1	43.30	43.30	POUCH
31	1	RA	OD6521660417-TOY56	1	44.82	44.82	UPSEX
34	1	GA	NH90133414-YFU2	1	16.04	16.04	LITBR
35	2	GA	NH90133414-YFU2	1	16.04	16.04	LITBR
36	3	GA	NH90133414-YFU2	1	16.04	16.04	LITBR

More Detailed Information

DJRowSet is both more flexible and more convenient for storing record images than arrays. Unlike arrays, each row in a row set can have a different number of elements as shown in this sample. In effect, the row set is like a jagged array.

Another advantage of row sets over arrays is that row sets have operations for inserting, appending, and deleting rows. With arrays, those operations require user-defined code.

Reference

See “DJRowSet Object Type” in the *Rapid Integration Flow Language Reference.*

See “Transformation Event Handlers” in the *Intermediate and Advanced Mapping User’s Guide*.

transformation

Dynamic SQL Lookup

17

Sometimes it is desirable or necessary to retrieve data that changes frequently. In such cases, dynamic SQL lookups serve this purpose.

Objectives

Return lookup values from a table in an Access database to an ASCII Delimited target file.

Skill Level

Advanced

Skill Set and Experience

- SQL
- RIFL Scripting
- Transformation Properties
- **DJImport** Variable

Design Considerations

This transformation sets a variable that is a connection to an Access 97 database. Care must be taken to test the connect string prior to production, especially if the transformation will be deployed in locations other than where it is written.

Sample Map Location

SamplesDir`\Transformations\Dynamic SQL Lookup.map.xml`

Sample Repository Configuration

The samples use workspaces and repositories to access the sample files. You must define that repository before running a sample transformation or process. For more information, see “Define a Samples Repository” in the “About the Samples” section.

Source Information

Connection
Source Connector: **ASCII Delimited**
Source File/URI:
SamplesDir`\Data\src_tutor1.asc`

Description
For this sample transformation, we declared a **DJImport** variable. In

Transformation Events ▸ Before Transformation, we declared the import connector type and the connection string. In **Transformation Events ▸ After Transformation**, we cleared the **DJImport** variable. In addition, we wrote a SQL query as the field expression for the target **Last Name** field.

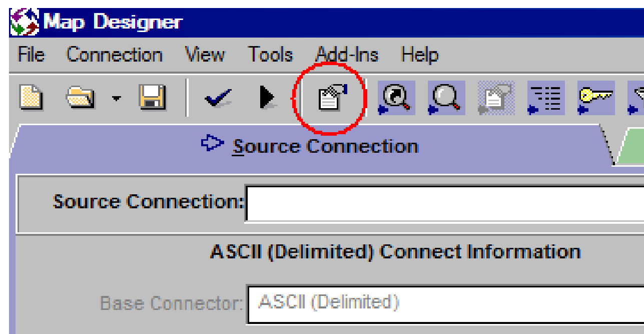
Target Information

Connection
Target Connector: **ASCII Delimited**
Output Mode: **Replace File/Table**
Target File/URI:
SamplesDir`\Data\trg_dynamic_sql_lookup.asc`

Procedure Explanation

To accomplish this transformation in Map Designer, the following was done:

1. We clicked the **Transformation and Map Properties** icon to open the window to define a global variable of type **DJImport** to use as an Access 97 connector.



2. In **Transformation Properties ▸ Transformation Events ▸ Before Transformation** we set an **Execute** action to declare the lookup connector type as Access 97 and instantiate the connection string to the **SamplesWork.mdb** Access 97 database:

   ' Declares the import spoke type.

   ```
   Set variablename = new djimport "Access 97"
   ```

   ' Sets the import connect string.

   ```
   variablename.connectstring =
   "database=Samples\Data\SamplesWork.mdb"
   ```

3 In the same **Transformation and Map Properties** window, we set an **After Transformation** event with an **Execute** action to clear the global variable:

```
Set variablename = Nothing
```

4 On the **Map** tab, we set a target field expression for the **Last Name** field. This is the expression that performs the dynamic SQL lookup:

```
variablename.sqlstatement = "Select [Last Name] from
Tutor1Date where [Account No]='" & Fields("Account
No") & "'"

variablename.fields("Last Name")
```

Note The dynamic SQL statement in this sample writes only the account number and last name to the target file.

Reference

See "Dynamic SQL Lookups" on page 9-2 in the *Intermediate and Advanced Mapping User's Guide*.

Dynamic SQL Lookup with Error Handling

18

Sometimes it is desirable or necessary to retrieve data that changes frequently. In such cases, dynamic SQL lookups serve this purpose. This sample includes error handling.

Objectives

Return lookup values from a table in an Access database to an ASCII Fixed target file and handle errors in a specified manner.

Skill Level

Advanced

Skill Set and Experience

- SQL
- RIFL Scripting
- Transformation Properties
- **DJImport** Variable

Design Considerations

This transformation sets a variable that is a connection to an Access 97 database. Care must be taken to test the connect string prior to production especially if the transformation will be deployed in locations other than where it is written.

Sample Map Location

SamplesDir`\Transformations\Dynamic SQL Lookup with Error Handling.map.xml`

Sample Repository Configuration

The samples use workspaces and repositories to access the sample files. You must define that repository before running a sample transformation or process. For more information, see "Define a Samples Repository" in the "About the Samples" section.

Source Information

Connection
Source Connector: **ASCII (Delimited)**
Source File/URI:
SamplesDir`\Data\src_dsqlreject.asc`

Description
For this sample transformation, we declared a **DJImport** variable. In

Transformation Events ▸ Before Transformation, we declared the import connector type and the connection string. In **Transformation Events ▸ After Transformation**, we cleared the **DJImport** variable. In addition, we wrote a SQL query as the field expression for the target **First Name** field. This expression contains the error handling code.

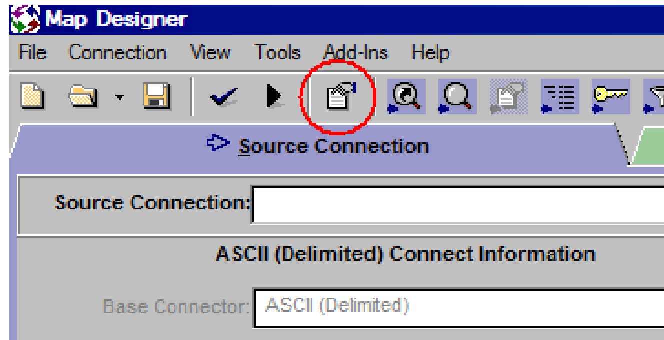
Target Information

Connection
Target Connector: **ASCII (Fixed)**
Output Mode: **Append to File/Table**
Target File/URI:
SamplesDir`\Data\trg_dsqldefault.asc`

Procedure Explanation

To accomplish this transformation in Map Designer, the following was done:

1. We clicked the **Transformation and Map Properties** icon to define a global variable of type **DJImport** to use as an Access 97 connector.



2. In **Transformation Properties ▸ Transformation Events ▸ Before Transformation,** we set an **Execute** action to declare the connector type as Access 97 and instantiate the connection string to the **SamplesWork.mdb** Access 97 database:

   ' Declares the import spoke type.

   ```
   Set variablename = new djimport "Access 97"
   ```

   ' Sets the import connect string.

   ```
   variablename.connectstring =
   "database=Samples\Data\SamplesWork.mdb"
   ```

3. We then set an **After Transformation** event with an **Execute** action to clear the global variable:

```
Set variablename = Nothing
```

4. On the **Map** tab, we set a target field expression for the **First Name** field. This is the expression that performs the dynamic SQL lookup. We included error handling in this target field expression:

'This expression traps any errors.

```
On Error GoTo Handle
```

' Executes the DSQLLookup.

```
variablename.sqlstatement = "Select [First Name]
from Tutor1Date where [Account No]='" &
Fields("Account No") & "'"
Return variablename.fields("First Name")
```

' This expression determines how to handle any errors.

```
Handle:
   Return Fields("First Name")
   Resume
```

More Detailed Information

When the transformation encounters an error, it drops to **ErrorHandler**, writes only the **First Name** field value to the target, then resumes the transformation.

Reference

See "Dynamic SQL Lookups" on page 9-2 in the *Intermediate and Advanced Mapping User's Guide* available with your product.

Search for the words "transformation event handlers" in the online documentation.

Dynamic SQL Lookup with Reject Records Handling

19

Sometimes it is desirable or necessary to retrieve data that changes frequently. In such cases, Dynamic SQL Lookups serve this purpose. This sample includes a method of handling rejected records.

Objectives

Return lookup values from a table in an Access database to a target file and handle rejected records in a specified manner.

Skill Level

Advanced

Skill Set and Experience

- SQL
- RIFL Scripting
- Transformation Properties
- Event Handlers, specifically the **OnError** Event
- Event Order of Precedence
- Basic understanding of reject record handling
- **DJImport** Variable

Design Considerations

This transformation implements a dynamic SQL lookup and writes records that do not have a match in the lookup table to a reject file. This implements the transformation level **OnError** event for error trapping.

For this sample transformation, we declared a **DJImport** variable. In **Transformation Events** ▸ **Before Transformation**, we declared the import connector type and the connection string. In **Transformation Events** ▸ **After Transformation**, we cleared the **DJImport** variable. In addition, we wrote a SQL query, our dynamic lookup, as the field expression for the target **First Name** field.

Sample Map Location

SamplesDir`\Transformations\Dynamic SQL Lookup with Rejects.map.xml`

Sample Repository Configuration

The samples use workspaces and repositories to access the sample files. You must define that repository before running a sample transformation or process. For more information, see “Define a Samples Repository” in the “About the Samples” section.

Source Information

Connection
Source Connector: **ASCII (Delimited)**
Source File/URI:
SamplesDir`\Data\src_dsqlreject.asc`

Description
The source file, **src_dsqlreject.asc**, contains sample data we used to match against sample data in the target file to trigger our reject on error.

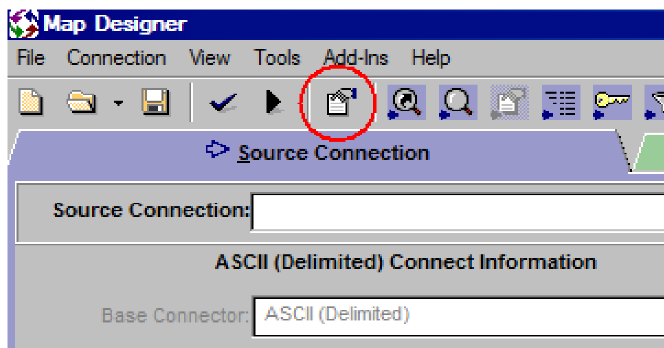
Target Information

Connection
Target Connector: **ASCII (Fixed)**
Output Mode: **Replace File/Table**
Target File/URI:
SamplesDir`\Data\trg_dsqlreject.asc`

Procedure Explanation

We used the following steps to accomplish this transformation in Map Designer:

1 In **Transformation and Map Properties** ▸ **Global Variables**, we set a **DJImport** variable as an Access 97 connector.



2 In **Transformation Properties** ▸ **Transformation Events** ▸ **Before Transformation** we set an **Execute** action to declare the connector type as Access 97 and instantiate the connection string to the Access 97 database named **SamplesWork.mdb**:

' Declares the import spoke type.

```
Set ImportVariable = new djimport "Access 97"
```

' Sets the import connect string.

```
ImportVariable.connectstring =
"database=Samples\Data\SamplesWork.mdb"
```

3 In **Transformation Properties** ▸ **RejectConnectionInfo**, we specified **ASCII (Delimited)** as the reject type for the reject file and built the following connect string:

```
codepage=ANSI;recordseparator=CRLF;
fieldseparator=,;fieldstartdelimiter='"';
fieldenddelimiter='"';header=False;
fielddelimitstyle=all;stripleadingblanks=True;
striptrailingblanks=True;maxdatalen=0;
File='Samples\Data\rej_dsqlreject.asc';
```

4 In **Transformation Properties** ▸ **Map Properties** ▸ **Global Variables**, we declared a variable named **ImportVariable** and selected **DJImport** data type. **ImportVariable** initializes the dynamic SQL lookup as a transformation global variable.

5 In **Map Properties** ▸ **Transformation Events** ▸ **BeforeTransformation**, we created an **Execute** action that sets the source connection type as Access 97 and sets the connect string to the Access database. We used the following code in this **BeforeTransformation** event:

'Declares the import connector type.

```
Set ImportVariable = new djimport "Access 97"
```

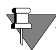
'Sets the import connect string.

```
ImportVariable.connectstring =
"Database=Samples\Data\SamplesWork.mdb"
```

6 In **Map Properties** ▸ **Transformation Events** ▸ **After Transformatio**n, we set an **Execute** action to clear the global variable:

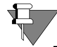
'This expression destroys the DJImport object. This is an important step as it frees the memory occupied by the DJImport object.

```
Set ImportVariable = Nothing
```



Note *Nothing* is a keyword that destroys the variable object.

7 In **Transformation Properties** > **Error Logging**, we set the following properties:



Note These are Transformation Properties. This means that they are intrinsically global in scope.

Error Logging Property	Value
Clear log file before each run	off (optional)
Flush log file to disk	off (optional)
Log Filename	djwin.log (default)
Fatal Errors	on (optional)
General Errors	on (optional)
Warnings	on (optional)
Informative Messages	on (optional)
Debug Messages	off (optional)
Break after error count of:	1 (optional)
Show first <#> fields of bad record	3 (optional)

8 On the **Map** tab, in the **Source** tree, we set an **AfterEveryRecord** event handler to execute the action **ClearMapPut Record**. This action clears the map buffer and writes the record to the target file.

9 In the **Target Field Expression** field for **First Name**, we wrote an expression that executes the dynamic SQL lookup. Following is the expression we entered for the **First Name** target field:

' Executes the DSQLLookup

```
ImportVariable.sqlstatement = "Select [First Name]
from Tutor1Date where [Account No]='" &
Fields("Account No") & "'"
```

```
ImportVariable.fields("First Name")
```

10 As the final step to construct our transformation, we set a target **OnError** event handler.

More Detailed Information

To view the target **OnError** event handler:

1. Navigate to the **Map** tab.
2. Find the **Target Record R1** tree in the lower left quadrant of your screen.
3. Locate R1 **Event Handlers** in the tree and expand the tree view.
4. Scroll to the **OnError** event handler near the bottom of the list of event handlers in the expanded tree view.
5. Click the ellipsis next to the ClearMapPut Record action. The **Actions and Parameters** dialog appears.

(ASCII (Fixed)) Targe

		Action Name	Action Comment
▶	1	ClearMapPut Record	
	2	Resume	
*			

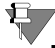
(ASCII (Fixed)) Target Record: R1, E

	Parameter Name	Required	Value
	target name	Yes	Reject
	record layout	Yes	R1
▶	count	No	
	counter variable	No	
	buffered	No	

Note that we set a **ClearMapPut Record Execute** action with a **Reject** parameter. This action fires when the **Reject Record** parameters are encountered. The **Reject Record** parameters are contained in the expression we wrote for the target R1 **First Name** field. Here is the expression again:

```
ImportVariable.sqlstatement = "Select [First Name]
from Tutor1Date where [Account No]='" &
Fields("Account No") & "'"
```

```
ImportVariable.fields("First Name")
```



Note We set a second **Execute** action, **Resume**, in the **OnError** event. If we had not set a **Resume Execute** action, the transformation would stop after it wrote the first record to the reject file.

When you design a transformation that writes to a reject records file, you must set a **Resume Execute** action to fire after a record is written to the reject records file.

Reference

See "Dynamic SQL Lookups" on page 9-2 in the *Intermediate and Advanced Mapping User's Guide*.

Search for the words "event precedence" and "transformation event handlers" in the online documentation.

See "Configuring Reject Connect Information" on page 2-15 in the *Intermediate and Advanced Mapping User's Guide*.

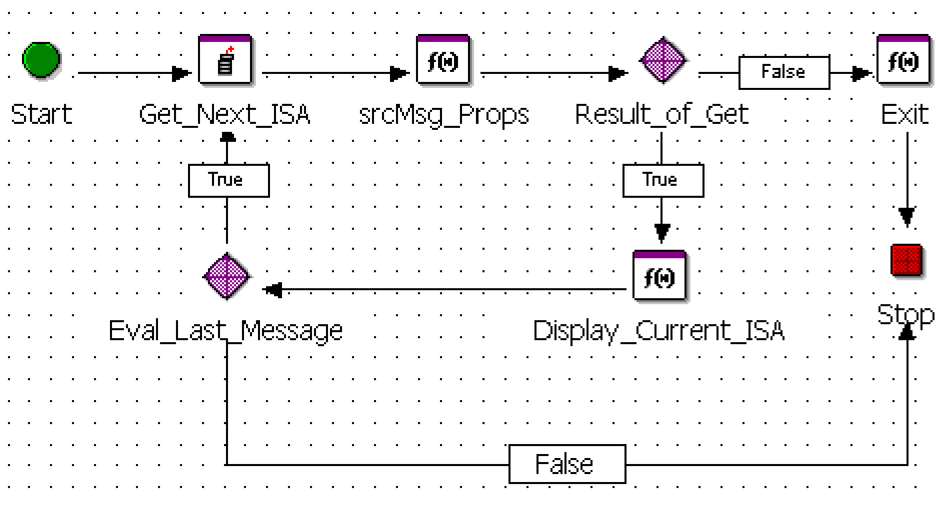
process

Using EDI X12 Iterator to Read Messages

20

The Process Designer Iterator function permits the breaking of a process into atomic units. In this sample we use iterators to segment an EDI document with multiple interchanges.

Objectives Break an EDI file into multiple interchanges to permit lookup of training partner information or to send responses for each interchange separately.



Skill Level Advanced

Skill Set and Experience

- Process Designer
- Map Designer
- RIFL Scripting
- Basic understanding of EDI file structure

Design Considerations EDI transactions must be processed by a specific map. This is easy to do using one of the other iterators. The Functional Group Iterator

(GS segment through GE segment) decomposes an interchange into sets of similar transactions so they can be routed to the appropriate map by branching in the process design.

Sample Process Location

SamplesDir`\Processes\Iterator_EDI_Batch.ip.xml`

Sample Repository Configuration

The samples use workspaces and repositories to access the sample files. You must define that repository before running a sample transformation or process. For more information, see "Define a Samples Repository" in the "About the Samples" section.

Other Files Location

SamplesDir`\Data\EDIBatch_Good.edi`

Procedure Explanation

This process uses one iterator, three scripts, and two decisions between **Start** and **Stop**:

1 **Start**
This first step instantiates the **DJMessage** object for use during the process:

```
Set myMsg = New DJMessage "myMsg"
```

2 **Get_Next_ISA**
We set up the first instance of an iterator here. This instance is based on the **TestBatch** iterator and uses the **GetMessage** action to capture the message for the next interchange from the iterator (segments ISA-IEA) and sets the variable **myMsg** to contain that message.

Note Before setting up the first iterator instance (**Get_Next_ISA**), we created the **TestBatch** iterator to use the **EDIBatch_Good.edi** source file.

3 **srcMsg_Props**
We added this script to write some header information for the message that was extracted in the previous step.

a. We used **Dim** to declare variables **A**, **B**, **C**, and **CR**:

```
Dim A, B, C, CR
```

b. We defined **A** as the current iterator count, **B** as the total iterations at the time of processing, and **C** as the priority (where **1** equals the last iteration):

```
A = myMsg.Properties("DJFT PieceNumber")
B = myMsg.Properties("DJFT TotalPieces")
C = myMsg.Properties("DJFT Priority")
```

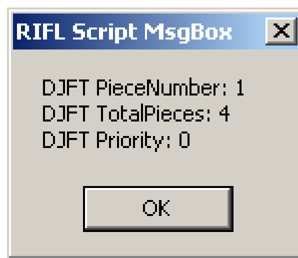
c. We defined **CR** to insert a carriage return (new line):

```
CR = Chr(13)
```

d. The final steps in this script cause a message box to open and display the message with some prefix text. If the priority variable (**C**) value is 1, we will also insert an additional message line indicating this is the last message received:

```
MsgBox("DJFT PieceNumber: " & A & CR & "DJFT
TotalPieces: " & B & CR & "DJFT Priority: " & C)

If C = 1 Then
MsgBox("Last Message Received")
End If
```

After the first message, the message box looks like this:



4 Result_of_Get

In this first decision, the script tests the iterator return code to ensure the message was retrieved:

```
Project("Get_Next_ISA").ReturnCode = 0
```

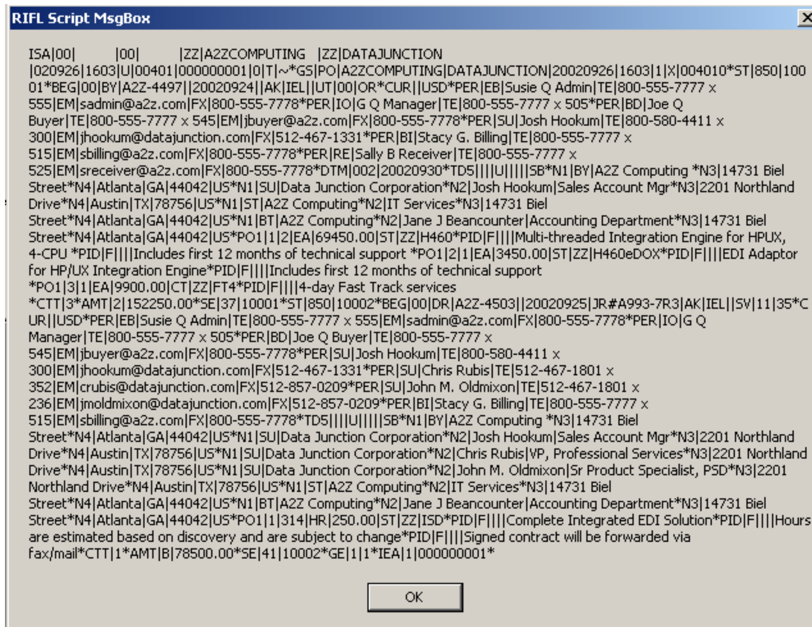
If the return code is zero (**0**), the process continues with the next step (**Display_Current_ISA**). If the return code is one (**1**), the process continues to the **Exit** step to terminate the process.

5 Display_Current_ISA

In this step we display the body of the current ISA with this expression:

```
MsgBox(myMsg.Body)
```

This results in the following display:



6 Eval_Last_Message

After each message is processed, we must determine if this was the last message or if other messages should be processed. We use the following expression to perform that test:

```
myMsg.Properties("DJFT Priority") = 0
```

If the result is zero (0), the process loops again for the next message. If the result is one (1), the process goes to the **Exit** step in preparation for ending the process.

7 Exit

When the final message has been processed, this step displays the Iterator step return code:

```
MsgBox("Iterator Step Return Code = " &
Project("Get_Next_ISA").ReturnCode)
```

8 Stop

The final step destroys the DJMessage object by setting the message variable to **Nothing**.

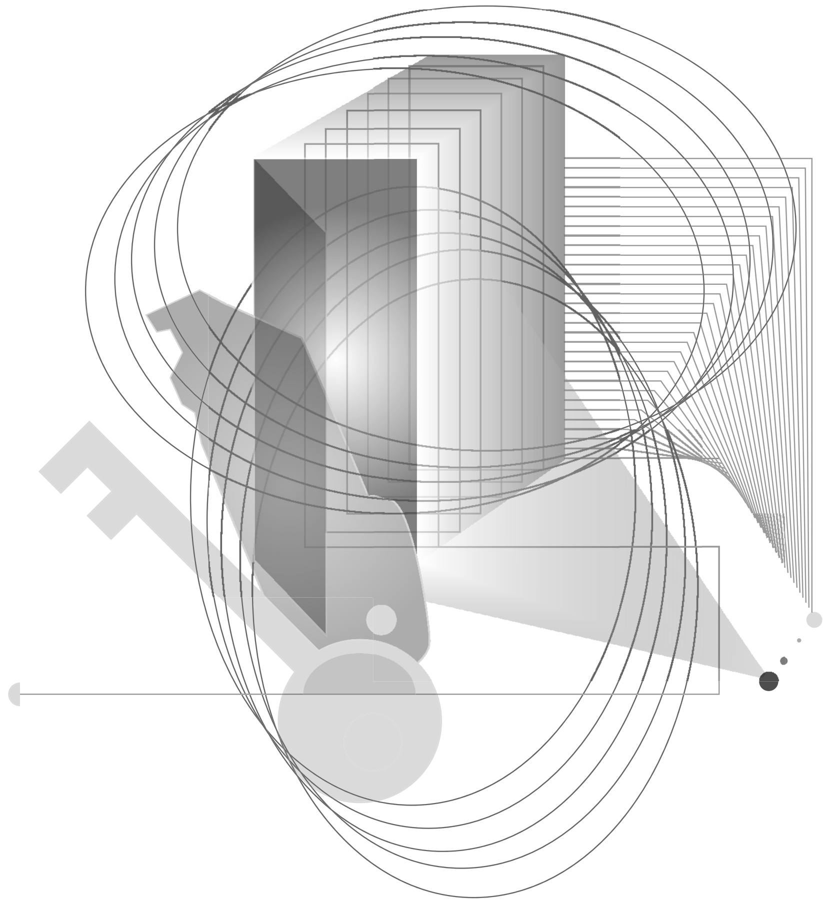
```
Set myMsg = Nothing
```

More Detailed Information

You can use the File Transfer Iterator to chunk files into smaller, more manageable units and send them to a destination. Each iterator has a respective *aggregator* (also known as *builder*) that is used on the receiving end to reassemble the file into a whole unit.

Reference

See “Sessions: Iterator Sessions” in the *Process Designer User’s Guide* available with your product.



Connector-Specific Samples

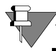
Microsoft Dynamics GP 10: Updating Records

22

Updating Parent and Child Records

Objectives

The Microsoft Dynamics GP 10 connector provides insert, update, and delete access by implementing a TargetRecordSet. The connector interacts with the GP web service, which provides access to the parent entities, such as Customer or Sales Order. The parent entities are transformed into multiple record types by Map Designer. You construct parent entities using record types when mapping the target. To manipulate the child entities, you must specify parent entities along with child entities.



Note You must have installed Data Integrator 9.2.0 or later to run this sample.

Skill Level

Intermediate

Skill Set and Experience

- Microsoft Dynamics GP 10
- Map Designer
- Event Actions

Sample Map Location

SamplesDir`\Transformations\Updatecustomer_gp.map.xml`

Sample Repository Configuration

The samples use workspaces and repositories to access the sample files. You must define that repository before running a sample transformation or process. For more information, see “Define a Samples Repository” in the “About the Samples” section.

Source Information

Source Connector: Null

Target Information

Target Connector: Microsoft Dynamics GP 10
Output Mode: Multiple

Before You Begin

➤ To redefine macros

1. You must redefine the macros in the transformation so that they point to your connection information for the **Web Service URL**, **Username**, and **Password** fields for the Microsoft Dynamics GP 10 target connector. For details on defining macros, see the topic "Macro Manager" in the *Getting Started Guide*.
2. Once you have defined your macros, return to the Target tab and select **Connect**.

Design Review

Updating parent record fields requires populating certain identity fields. To update child members, you must first specify the parents. In cases where the child record type exposes a _LineNumber field, the connector uses the field to lookup and alter the appropriate element in the child list. For others, another field may be required, such as a Key_Id.

➤ To update records

To update [Customer]Addresses1 and [Customer]Addresses2, we did the following:

1. Designed a map with Null as the source type, Microsoft Dynamics GP 10 as the target type, and connected to Microsoft Dynamics GP.
2. In a Source record event handler, we created an **AfterEveryRecord** event and added the following events:

Action	Parameters
ClearMap	{Target},{Customer1},{},{}
Update Record	{Target},{Customer1},{Customer},{},{}
ClearMap	{Target},{Customer}Addresses1,{},{}
Update Record	{Target},{Customer}Addresses1,{[Customer]Addresses},{},{}
ClearMap	{Target},{Customer}Addresses2,{},{}
Update Record	{Target},{Customer}Addresses2,{[Customer]Addresses},{},{}

3. Next, we cleared the map for the Customer1 record layout.
4. Then we specified an Update Record action for the Customer1 record layout and Customer record.
5. We cleared the map for the Customer Addresses1 record layout.
6. We used an Update Record action to update the Customer Addresses1 record layout and the Customer record.
7. We repeated steps 5 and 6 to update the Addresses2 record.
8. In the target pane, we mapped the Key_Id field for Customer.

Results

Run the map. Note that the parent record Customer1, the child records Customer Addresses1 and Customer_Addresses2, and the Key_ID for Customer Addresses are updated.

Reference

To learn how to create and delete child records, see the topic "Microsoft Dynamics GP 9 and 10" in the *Source and Target Connectors User's Guide.*

Microsoft Dynamics CRM 4.0: Inserting Records

23

Inserting Records, Designating a Primary Contact for the Account Entity

Objectives

Inserts Account and Contact records, then makes Contact the Primary Contact for the Account in a Microsoft Dynamics CRM 4.0 target entity.

Important Note

You must have installed Data Integrator 9.2.0 or later to open this sample, since the target connector is new in 9.2.0. This sample transformation was not designed to be run. Instead, this example teaches you to set up events and mapping so you can insert records into a Microsoft Dynamics CRM 4.0 target entity.

Skill Level

Intermediate

Skill Set and Experience

- Microsoft Dynamics CRM 4.0
- Map Designer
- Event Actions

Sample Map Location

SamplesDir`\Transformations\insert.account.contact.live.map.xml`

Sample Repository Configuration

The samples use workspaces and repositories to access the sample files. You must define that repository before running a sample transformation or process. For more information, see “Define a Samples Repository” in the “About the Samples” section.

Source Information

Source Connector: Null

Target Information

Target Connector: Microsoft Dynamics CRM 4.0
Output Mode: Multiple

Description
The target file is a web address. We changed the **ServiceType** property to Online. We also entered a path to the batch response file at **BatchResponse**.

Before You Begin

➢ To redefine macros

1. You must redefine the macros in the transformation to point to your connection information for the **Organization**, **User ID**, and **Password** fields for the Microsoft Dynamics CRM 4.0 target connector. If you want to use an On-Premise connection instead of the Online connection used in the sample, you must also define a macro for **Server**. Format is http://URL/domain, or https://URL/domain. For details on defining macros, see the topic "Macro Manager" in the *Getting Started Guide*.
2. Once you have defined your macros, return to the Target tab and select **Connect**.

Design Review

The following steps describe the parameters used in the **AfterEveryRecord** event handler on the source layout **R1**.

1. On the Map tab, we selected the **AfterEveryRecord** event handler.
2. Next we added a **ClearMapInsert Record** actions for the target record layouts account_1 and contact_1.
3. Then for the target record layout account_2, we added a **ClearMap** action and an **Update Record** action.

   On the target entity, three target records are mapped, account_1, contact1, and account_2. Select account_2 Fields and note that target field accountid is set as the primary key. The target field primarycontactid is the lookup field.

Results

None. See "Important Note" on page 23-1.

Reference

Search for the words "event actions" and "record type event handlers" in the online documentation.

"Microsoft Dynamics CRM 3.0 and 4.0" in the *Source and Target Connectors Guide*

Oracle Siebel CRM On Demand 14: Deleting Child Records

24

Deleting a Child Record From a Child Entity

Objectives

Removes a Account_Contact child entity from an Account child entity and shows how to use the Oracle Siebel CRM On Demand deletechild action.

Important Note

You must have installed Data Integrator 9.2.0 or later to open this sample, since the target connector is new in 9.2.0.

This sample transformation was not designed to be run. Instead, this example teaches you how to set up events and mapping so you can delete a child record from a child target entity.

Skill Level

Intermediate

Skill Set and Experience

- Oracle Siebel CRM On Demand 14
- Map Designer
- Event Actions

Sample Map Location

SamplesDir`\Transformations\DeleteChild_AccountContact.map.xml`

Sample Repository Configuration

The samples use workspaces and repositories to access the sample files. You must define that repository before running a sample transformation or process. For more information, see “Define a Samples Repository” in the “About the Samples” section.

Source Information

Source Connector: Null

Target Information

Target Connector: Oracle Siebel CRM On Demand 14
Output Mode: Multiple
Target File/URI: secure-ausomxapa.crmondemand.com

Description

The target file is a web address and the target properties require no changes.

➢ To redefine macros

1. You must redefine the macros in the transformation so that they point to your connection information for the Server, User ID, and Password fields for the Oracle Siebel CRM On Demand 14 target connector. For details on defining macros, see the topic "Macro Manager" in the *Getting Started Guide*.

Once you have defined your macros, return to the Target tab and select **Connect**.

Design Review

The following steps describe the parameters used in the **AfterEveryRecord** event handler on the source **R1** table.

1. First we selected the **AfterEveryRecord** event handler.
2. Next we added a **ClearMap** action for the target record layout Account and add account.child as the count parameter.
3. Then we added a **Delete Record** action to delete the account.child entity from the Account entity.
4. We added a **ClearMap** action for the target record layout Account_Contact and added account.contact.child as the count parameter.
5. Next we added a **Delete Record** action to delete the account.contact.child record from the Account_Contact target record layout.

Then we mapped the accountid for both the Account and Account_Contact entities.

1. In the target field expression, for the Account target record, we typed the account ID "AAPA-1CZDFR".
2. For the Account_Contact target record, we mapped the accountid field to retreive the account ID number.

```
Targets(0).Records("Account").Fields("accountid")
```

3 In the target field expression, for the Account_Contact target record, we typed the contact ID “AAPA-1CZDQX”.

Results

None. See “Important Note” on page 24-1.

Reference

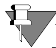
Search for the words “event actions” and “record type event handlers” in the online documentation.

Netsuite 2.6: Entering Sales Orders

25

Objectives

The Netsuite 2.6 connector provides connection to Netsuite entities. This example inserts sales orders from XML source opportunity fields into the Netsuite target entity fields.



Note You must have installed Data Integrator 9.2.0 or later to run this sample.

Skill Level

Intermediate to Advanced

Skill Set and Experience

- Netsuite 2.6
- Map Designer
- Event Actions
- Familiar with multiple record types

Sample Map Location

SamplesDir`\Transformations\EnterSalesOrderFromOpportunity.map.xml`

Sample Repository Configuration

The samples use workspaces and repositories to access the sample files. You must define that repository before running a sample transformation or process. For more information, see "Define a Samples Repository" in the "About the Samples" section.

Source Information

Source Connector: XML

Target Information

Target Connector: Netsuite 2.6
Output Mode: Multiple

Using Macros to Connect

➢ To redefine macros

1. You must redefine the macros in the transformation so that they point to your connection information for the **Account** and **Email** fields for the Netsuite 2.6 target connector. For details on defining macros, see the topic "Macro Manager" in the *Getting Started Guide*. Alternatively, you can delete the macros and type in your connection information.
2. Once you have defined the macros, return to the Target tab and select **Connect** to test the connection.

Property Options

Set the Netsuite target property **ShowChildren** to True.

Set the Netsuite target property **BatchResponse** to generate an .xml file to help you troubleshoot issues. Example: C:\nsaddress.xml

Note The BatchResponse property is especially important because errors are usually written to the batch response file instead of to the Pervasive log file.

Log File Location

Currently, the sample transformation points to the log file here:

C:\Documents and Settings\username\Pervasive\Logs\MapDesigner\TransformMap.log

To change the log file path, select **View** > **Transformation and Map Properties** > **Error logging** > **Log Filename**. Browse to your TransformMap.log file. Click **OK**.

Design Review

Inserting parent record fields requires populating certain identity fields. To insert child members, you must first specify the parents.

➢ To insert sales orders

To insert sales orders from the Opportunity entity, we did the following:

1. Designed a map with XML as the source type, and Netsuite 2.6 as the target type.
2. Modified the target connection options as shown in "Using Macros to Connect".

3 In a Source record event handler, we created an **AfterEveryRecord** event and added the following events:

Record Name	Event Name	Action	Parameters
Items	AfterEveryRecord	ClearMapInsert	{Target} {SOItems}{SalesOrder Item}{}{}
Opportunity	AfterEveryRecord	ClearMapInsert	{Target} {SalesOrder}{SalesOrder}{}{}

Required Design Components

To insert sales orders from Opportunity source fields to Netsuite entities, you must do the following:

- Associate Opportunity fields with SalesOrder fields.
- Include an internal or external ID from the Customer fields into the SalesOrder fields. According to Netsuite, each record is uniquely identified by its record type with a combination of the following:
    - a system-generated NetSuite internal ID
    - an externalID that is provided during an update or at the time of record creation

  You can hard-code the external ID, or as an example, you can create an external ID by mapping a customer's last and first names from the Source. Note that customer is referred to as "entity" in Netsuite. So in the SalesOrder fields, we included Entity_InternalId as the customer internal ID. For more information on external and internal IDs, see netsuite.com and search for the key words "external ID" and "internal ID".
- Include a Transaction ID in the target SalesOrder fields. In the sample, this ID is listed as "TranId".
- Include a sales representative internal ID. In the SalesOrder fields, it is listed as "SalesRep_InternalId".
- Include an Opportunity external or internal ID. In the SalesOrder fields, it is listed as "Opportunity_ExternalId".

Caution If you do not include each of the items listed above in your mapping, errors are returned to your batch response file. For the location of the file, see "Property Options".

Results

Run the map. Note that sales order data from the Opportunity fields in the XML source are inserted into the Netsuite target fields.

Reference

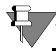
For more information on the Netsuite 2.6 connector, see the "Netsuite 2.6 Connector topic" in the *Source and Target Connectors User's Guide.*

Netsuite 2.6: Adding Addresses to Addressbook

26

Objectives

The Netsuite 2.6 connector provides connection to Netsuite entities. To manipulate the child entities, you must specify parent entities along with child entities. This example adds Salesforce.com source addresses to the Netsuite address book target entity.



Note You must have installed Data Integrator 9.2.0 or later to run this sample.

Skill Level

Intermediate to Advanced

Skill Set and Experience

- Netsuite 2.6
- Map Designer
- Event Actions
- Familiar with multiple record types

Sample Map Location

SamplesDir`\Transformations\InsertAddressesAddressBook.map.xml`

Sample Repository Configuration

The samples use workspaces and repositories to access the sample files. You must define that repository before running a sample transformation or process. For more information, see “Define a Samples Repository” in the “About the Samples” section.

Source Information

Source Connector: Salesforce 10.0

Target Information

Target Connector: Netsuite 2.6
Output Mode: Multiple

Using Macros to Connect

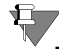
➢ **To redefine macros**

1. You must redefine the macros in the transformation so that they point to your connection information for the **Account** and **Email** fields for the Netsuite 2.6 target connector. For details on defining macros, see the topic “Macro Manager” in the *Getting Started Guide*. Alternatively, you can delete the macros and type in your connection information.
2. Once you have defined your macros, return to the Target tab and select **Connect** to test the connection.

Property Options

Set the Netsuite target property **ShowChildren** to True.

Set the Netsuite target property **BatchResponse** to generate an .xml file to help you troubleshoot issues. Example: C:\myresponse.xml



Note The BatchResponse property is especially important because errors are usually written to the batch response file instead of to the Pervasive log file.

Log File Location

Currently, the sample transformation points to the log file here:

C:\Documents and Settings\username\Pervasive\Logs\MapDesigner\TransformMap.log. To change the log file path, select **View** > **Transformation and Map Properties** > **Error logging** > **Log Filename**. Browse to your TransformMap.log file. Click **OK**.

Design Review

Inserting parent record fields requires populating certain identity fields. To insert child members, you must first specify the parents.

➢ **To add Salesforce addresses to the Netsuite addressbook entity**

1. Design a map with Salesforce 10.0 as the source type. Modify the source connection options as shown in “Using Macros to Connect”.
2. Create a query statement. In the sample, we created a query to select only Pervasive Software accounts.
3. Select Netsuite 2.6 as the target type.

4 Modify the target connection options as shown in “Using Macros to Connect”.

5 In a Source record event handler, create an **AfterEveryRecord** event and add the following events:

Record Name	Event Name	Action	Parameters
SFDC	AfterEveryRecord	ClearMapInsert	{Target}{Customer}{Customer}{}{}
SFDC	AfterEveryRecord	ClearMapInsert	{Target}{Address}{Customer Addressbook}{2}{cnt}

6 Create a counter variable to loop twice and to map two different sets of addresses from the source to the target.

7 Note that in the target Customer fields, we added an ExternalId field. The hard-coded expression is “SFDC2NS”. You can also create an external ID by mapping a customer’s last and first names from the source data. For more information on external and internal IDs, see netsuite.com and search for the key words “external ID” and “internal ID”.

8 Next, we must create metadata expressions to map billing and and shipping data from Salesforce into address fields in the Netsuite target. From the source, we are mapping BillingStreet, BillingCity, BillingState, BillingPostalCode, and BillingCountry records into one addressbook record in Netsuite. We do the same for ShippingStreet and the other shipping address fields.

At Addr1, we include this expression:

```
Sources(0)Fields(8+(5*(cnt-1)))
```

We use “8” because BillingStreet is at the 8th index in the source entity. The number “5” is added because there are five fields between BillingStreet and ShippingStreet. Then we subtract 1 from the cnt counter variable to begin at the first iteration and move through the fields. For more information on creating metadata expressions, search for the keywords “transformation metadata” in the online help.

Results

Run the map. Note that the addresses from the Salesforce source now appear in the Netsuite customer addressbook target records.

Reference

For more information on the Netsuite 2.6 connector, see the "Netsuite 2.6 Connector topic" in the *Source and Target Connectors User's Guide.*

Converting an EDI Source Containing an N1 Loop

27

Objectives

The sample transformation converts an EDI source file that contains an N1 loop into an ASCII Delimited target file. The source file uses the 4010 standard. If you are using the 5010 standard, you will also find the sample useful.

Skill Level

Advanced

Skill Set and Experience

- Map Designer
- EDI file structure
- Event Handlers

Sample Map Location

SamplesDir`\Transformations\`EDI_src_N1_Loop.map.xml

Sample Repository Configuration

The samples use workspaces and repositories to access the sample files. You must define that repository before running a sample transformation or process. For more information, see "Define a Samples Repository" in the "About the Samples" section.

Source Information

Connection
Source Connector: EDI x12
Document Schema: EDI_N1_Loop_x12.4010.108.0.0.ds.xml
Source File: EDI_N1_Loop.edi

Target Information

Connection
Target Connector: ASCII Delimited
Output Mode: Replace
Target File/URI: EDI_N1_Loop.txt

Before You Begin

➢ To set up transformation before running

Before you run the sample transformation and view the results, you must complete the following steps:

1. From the Tools menu, select **Define Macros** and create a new macro named EDI_Src_N1Loop whose value is the path to the source file.
2. Click **Connect** to connect to the source file.
3. From the toolbar, open the Source Data Browser to browse the source data.
4. Next to Document Schema, click the ellipsis to open the document schema file. Review the contents.
5. Click **Connect** to connect to the target file.

Review

Transformation Details

The following table outlines how the transformation was created.

Name	Location	Option Selected	Remarks
EDI_N1_Loop.edi	Source Connection tab	Source file	EDI (XM12) source
EDI_N1_Loop_x12.4010.108.0.0.ds.xml	EDI (X12) Properties	SchemaFile	Document schema file to provide structure for source file.
EDI_N1_Loop.txt	Target Connection tab	Target file	Header property is set to True. This target file is created after the transformation is run.
N_Count CMPCnt	Transformation and Map Properties	Global Variables	Two global variables called N_Count and CMPCnt with initial values of 0.

Transformation Design

Below, we provide details on the mapping steps.

1. In Transformation and Map Properties in the BeforeTransformation event, we set an Execute Action to define a Global N_Array[14,4]. Alternatively, you can specify N_Array as a global variable. For clarity, the zero position of the array is not used in this sample.
2. Select the Map All tab. If the Map Fields tab is displayed, click the Map Fields tab, then click **Map All** in the main toolbar.
3. In the Source tree, we expanded Record Types, then expanded the L.N1record type. Then we expanded [L.N1]N1 Event Handlers and clicked **AfterEveryRecord**. We created an Execute action and an If Then Else statement.

Open the script to review what was done. In this sample, the array position depends upon whether it is Ship To or Remit To. For each N1 Loop, data is stored in an array, so if the field value of N1_01_98 is ST, it increments the counter N_Count and stores the company value in N_Array as the Ship To name. If the value is not ST, the company name is the Remit To name. The script assigns all other N1 loop values to N_Array, checking each time whether or not field N1_01_98 is ST and assigning either Ship To or Remit To.

4. Next, we expanded [L.N1]N2 Event Handlers and clicked **AfterEveryRecord**. Then we created an Execute action to include an If Then Else statement that added the first and last names to the array.

   We used the same procedure for [L.N1]N3 that adds the address fields and [L.N1]N4 that adds the city, state, and zip fields.

5. Then we expanded the SE record and SE Event Handers, then clicked **AfterEveryRecord**. In a ClearMapPut Record action, we specified the N_Count variable as the count value and the CMPCnt as the counter variable value. The count parameter specifies how many times this action fires and the counter variable parameter contains the iteration number.

6. Next, we moved to the target. In the Target Field expressions, we created RemitToName and SupplierName target field names and assigned specific values from the N_Array.

7. Save the transformation.

Results

Click **Validate Map** in the toolbar to validate the map and fix any validation errors. Run the transformation. The target file EDI_N1_Loop.txt is created. Click the Target Data Browser to browse the target data. Two records are returned that include the SupplierCompany AAL Capital Management and Novus Health Group company information.

Reuse Notes

Save the transformation as a new name in your workspace. You can use the transformation as a template each time you want to transform EDI data with an N1 Loop into ASCII Delimited text. This sample included ST segments. To reuse this transformation, you can substitute your data in place of the ST segments.

Reference See EDI X12 or EDI/EDIFact in web help.

Index

A

actions
- buffered Put Tree 12-1

Advanced Skill Level
- Aggregating Records 13-1, 24-1
- Complex Date Filtering 15-1
- Dynamic SQL Lookup 17-1
- Dynamic SQL Lookup with Error Handling 18-1
- Dynamic SQL Lookup with Reject Records Handling 19-1
- Manipulating Binary Dates at the Bit Level 14-1
- Using an Exe Step to Run Multiple Transformations 7-1
- Using EDI X12 Batch Transaction Iterator to Read Messages 20-1
- Using the FileList Function in a Process 9-1
- Working with DJRowSet and Arrays 16-1

aggregating records 13-1
arrays 16-1

B

Basic Skill Level
- Filtering Source Data 3-1
- Sorting Source Data 4-1
- Standardizing Multiple Date Formats 5-1
- Using Conditional Put Actions with Event Handlers 2-1
- Using DJX to Pass Variables to a SQL Stored Procedure 6-1

Binary 14-2
bit-level manipulation of binary dates 14-1
buffered Put Tree action 12-1

C

checking for existence of files 9-2
ClearInitialize action 13-3
Complex Date Filtering 15-1
conditional put record 2-1

D

data
- filtering 3-1
- restricting 3-1
- sorting 4-1, 11-1

date filtering, complex 15-1
date formats
- standardizing 5-1

date formats, standardizing 5-1
DJImport 17-1, 18-1
DJMessage 20-2
DJRowSet 16-1
DJX 6-1
Dynamic SQL Lookup 17-1
Dynamic SQL Lookup with Error Handling 18-1
Dynamic SQL Lookup with Reject Records Handling 19-1

E

EDI X12 Functional Group Iterator 20-1
Electronic Data Interchange (EDI) 10-1
error handlers 18-1
event handlers 2-1, 13-1, 24-1
- OnDataChange 11-1

F

File Transfer Iterator 20-5
FileList 9-1
files
- checking for existence 9-2

filtering data 3-1

G

global variables 8-1

H

hierarchical records, mapping 12-1

I

Intermediate Skill Level

Mapping Database Records to EDI 10-1
Setting OnDataChange Events 11-1
Using Buffered Put Tree to Create Hierarchical Records 12-1
Using Global Variables in Transformations 8-1
Iterator and EDI Batch Splitter 20-1
Iterators
EDI X12 Functional Group 20-1
File Transfer 20-5

L

line breaks in sample code 13-2
log message 2-1, 8-1

M

Manipulating Binary Dates at the Bit Level 14-1
mapping hierarchical records 12-1

O

OnDataChange events 13-4, 16-1
order of precedence 13-1, 24-1

P

passing variables to SQL 6-1
precedence, order of 13-1, 24-1
Processes
Using DJX to Pass Variables to a SQL Stored Procedure 6-1
Using EDI X12 Batch Transaction Iterator to Read Messages 20-1
Using the FileList Function in a Process 9-1
Put Tree action 12-1

R

record aggregation 13-1, 24-1
restricting data 3-1
RIFL scripting
aggregating records 13-1, 24-1
conditional put actions 2-1
date filtering 15-1
dynamic SQL lookup 17-1
with error handling 18-1
EDI X12 batch transaction iterator 20-1
filtering source data 3-1
global variables 8-1
passing variables to SQL 6-2

S

sample code line breaks 13-2
SamplesDir
defined 0-vi
Setting OnDataChange Events 11-1
sorting data 4-1, 11-1
Source Connection
27-1
Access 97 10-1
ASCII (Delimited) 3-1, 4-1, 5-1, 8-2, 11-1, 12-2, 13-1, 15-1, 16-1, 17-1, 18-1, 19-2
ASCII (Fixed) 2-1
binary 14-1
database 10-1
text 2-1, 3-1, 4-1, 8-2, 11-1, 12-2, 13-1, 15-1, 16-1, 17-1, 18-1, 19-2, 27-1
SQL Lookup, dynamic 17-1, 18-1, 19-1
SQL, passed variables 6-1

T

Target Connection
1-1, 27-1
ASCII (Delimited) 3-2, 4-2, 5-2, 15-2, 17-2, 18-2, 19-2
ASCII (Fixed) 2-1, 8-2, 13-2
dBASE IV 14-1
EDI (X12) 10-2
Excel 97 11-2
text 1-1, 2-1, 3-2, 4-2, 8-2, 13-2, 17-2, 18-2, 19-2, 27-1
XML 12-2, 16-2
Transformations
Aggregating Records 13-1, 24-1
Complex Date Filtering 15-1
Dynamic SQL Lookup 17-1
Dynamic SQL Lookup with Error Handling 18-1
Dynamic SQL Lookup with Reject Records Handling 19-1
Filtering Source Data 3-1
Mapping Database Records to EDI 10-1
Setting OnDataChange Events 11-1
Sorting Source Data 4-1
Standardizing Multiple Date Formats 5-1

Using Buffered Put Tree to Create Hierarchical Records 12-1
Using Conditional Put Actions with Event Handlers 2-1
Using Global Variables in Transformations 8-1
Working with DJRowSet and Arrays 16-1

U

Using DJX to Pass Variables to a SQL Stored Procedure 6-1

V

variables, global 8-1

X

XML target 12-2, 16-2