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
- Preparing to Use the Samples
- Finding Useful Samples
- Skill and Experience Requirements
- Programming Samples
- Command Line Continuation

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

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

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

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

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
Using Macros to Change Connections

**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), UserID $(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

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)
Using Conditional Put Actions with Event Handlers

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:

   ```vba
   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:

   ```vba
   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:

   ```vba
   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:

   ```vba
   End if
   ```
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**


Also see “If...Then...Else Statement” in the Rapid Integration Flow Language Reference.
Using Conditional Put Actions with Event Handlers
Filtering Source Data

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**

Source Connector: **ASCII (Delimited)**
Filtering Source Data

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 drag 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:

\[
\text{Records("R1").Fields("State") == "TX"}
\]

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

\[\text{Note}\]

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

Filtering Source Data
Sorting Source Data

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)
Sorting Source Data

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

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

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.
Sorting Source Data
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.
Standardizing Multiple Date 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.

   ![Source Data Browse](image)

<table>
<thead>
<tr>
<th>Record No</th>
<th>Field1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1/1/44</td>
</tr>
<tr>
<td>2</td>
<td>02/2/98</td>
</tr>
<tr>
<td>3</td>
<td>3/3/78</td>
</tr>
<tr>
<td>4</td>
<td>4/30/01</td>
</tr>
<tr>
<td>5</td>
<td>5/31/02</td>
</tr>
<tr>
<td>6</td>
<td>06/6/00</td>
</tr>
<tr>
<td>7</td>
<td>7/7/78</td>
</tr>
<tr>
<td>8</td>
<td>08/31/00</td>
</tr>
<tr>
<td>9</td>
<td>9/30/40</td>
</tr>
<tr>
<td>10</td>
<td>10/31/55</td>
</tr>
<tr>
<td>11</td>
<td>11/30/01</td>
</tr>
<tr>
<td>12</td>
<td>12/31/03</td>
</tr>
<tr>
<td>13</td>
<td>013002</td>
</tr>
</tbody>
</table>

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")
   ```

5-2
The next expression evaluates the source date and calls the specific `DateValMask` expression to convert the date to the desired format.

```vbnet
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:

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

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:

```vbnet
DateValMask(Fields("sourcedate"), "m/d/YY1900").
```
Standardizing Multiple Date Formats

**Reference**

Using DJX to Pass Variables to a SQL Stored Procedure

In this sample, we want 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
Using DJX to Pass Variables to a SQL Stored Procedure

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:
     
```plaintext
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:
     
```plaintext
Param1 = "10019"
Exec1 = "Delete from Tutor1 Where [Account No] = '10019'"
msgbox(Ejec1)
```

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:

```plaintext
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:

```plaintext
Param1 = ""
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:
Using DJX to Pass Variables to a SQL Stored Procedure

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.
Using an Exe Step to Run Multiple Transformations

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

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

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

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Variable Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connector</td>
<td>DJImport</td>
</tr>
<tr>
<td>DataFilesInfo</td>
<td>DJRowSet</td>
</tr>
<tr>
<td>recordCounter</td>
<td>Variant (initial value is 1)</td>
</tr>
</tbody>
</table>

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**

Using an Exe Step to Run Multiple Transformations
Using Global Variables in Transformations

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.

<table>
<thead>
<tr>
<th>Scope</th>
<th>Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dim</td>
<td>Dim</td>
</tr>
<tr>
<td></td>
<td>Local to Script Module</td>
</tr>
<tr>
<td>Global</td>
<td>Private</td>
</tr>
<tr>
<td></td>
<td>Throughout Map Design</td>
</tr>
<tr>
<td>Global</td>
<td>Public</td>
</tr>
<tr>
<td></td>
<td>Throughout Process Design</td>
</tr>
</tbody>
</table>
Using Global Variables in Transformations

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 ClearMapPutRecord 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 "dddddd" style (using default short style of m/d/yy):
      
      ```
      If IsDate(A) then
          Format(A,"dddddd")
      ```

   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.
Using Global Variables in Transformations

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: [xml:db: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: [xml:db: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

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
Using the FileList Function in a Process

**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.
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 **getFileDialog** and entered the following RIFL expression:

   ```rifl
   '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 **getFileDialog** step into a temporary file named **src_sourcefile.asc**.
Using the FileList Function in a Process

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.

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

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

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

' Increment the File_Count variable

```vbp
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:

```vbp
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:

```vbp
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.
Using the FileList Function in a Process
Mapping Database Records to EDI

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.
<table>
<thead>
<tr>
<th>Target Information</th>
<th>Connection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target Connector:</td>
<td>EDI (X12)</td>
</tr>
<tr>
<td>Target Schema:</td>
<td>ControlSegments.ds.xml</td>
</tr>
<tr>
<td>Output Mode:</td>
<td>Replace File/Table</td>
</tr>
<tr>
<td>Target File/URI:</td>
<td>SamplesDir\Data\ControlSegments.edi</td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>Source Fields</th>
<th>Target Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>subNameFirst</td>
<td>NM1_04_1036</td>
</tr>
<tr>
<td>subNameMiddle</td>
<td>NM1_05_1037</td>
</tr>
<tr>
<td>subLastOrgName</td>
<td>NM1_03_1035</td>
</tr>
</tbody>
</table>

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/
Mapping Database Records to EDI

Reference
See “EDI (X12)” in the Source and Target Connectors User’s Guide available with your product.
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
Source Connector: ASCII (Delimited)
Source File/URI:
Part 1: Sorted Source
SamplesDir\Data\Accounts_SortedbyState.txt
Setting OnDataChange Events

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.

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"
   
   \[
   \text{varState} = \text{Records("R1")}.\text{Fields("State")}
   \]

   ' Increment the counter for the number or records within this block
   
   \[
   \text{varCounter} = \text{varCounter} + 1
   \]

   ' Accumulate the balance for the records within this block
   
   \[
   \text{varBalance} = \text{varBalance} + \text{Records("R1")}.\text{Fields("Balance")}
   \]

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

<table>
<thead>
<tr>
<th>State</th>
<th>Zip</th>
<th>Email</th>
<th>Birth Date</th>
<th>Favorites</th>
<th>Standard</th>
<th>Payments</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>AK</td>
<td>99775-6480</td>
<td><a href="mailto:flibia@air.com">flibia@air.com</a></td>
<td>07/02/1945</td>
<td>GA</td>
<td>112.00</td>
<td>120.85</td>
<td>120.85</td>
</tr>
<tr>
<td>AK</td>
<td>99901-0001</td>
<td><a href="mailto:robert@desk.com">robert@desk.com</a></td>
<td>03/22/1942</td>
<td>CA</td>
<td>121.00</td>
<td>121.00</td>
<td>212.12</td>
</tr>
<tr>
<td>AL</td>
<td>35254-1176</td>
<td><a href="mailto:areeg@permail.com">areeg@permail.com</a></td>
<td>11/30/1967</td>
<td>KA</td>
<td>117.00</td>
<td>117.00</td>
<td>170.86</td>
</tr>
<tr>
<td>AR</td>
<td>71601</td>
<td><a href="mailto:widet@mon.com">widet@mon.com</a></td>
<td>07/03/1957</td>
<td>SE/FL</td>
<td>162.00</td>
<td>400.00</td>
<td>627.01</td>
</tr>
<tr>
<td>AZ</td>
<td>85224-4853</td>
<td><a href="mailto:swilliams@vaharelec.com">swilliams@vaharelec.com</a></td>
<td>12/13/1965</td>
<td>PA</td>
<td>150.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>AZ</td>
<td>85224-3867</td>
<td><a href="mailto:scott@vcmail.net">scott@vcmail.net</a></td>
<td>10/17/1959</td>
<td>CO</td>
<td>393.00</td>
<td>200.00</td>
<td>393.79</td>
</tr>
<tr>
<td>AZ</td>
<td>85013-2332</td>
<td><a href="mailto:tonw55@pcmail.net">tonw55@pcmail.net</a></td>
<td>12/01/1967</td>
<td>CA/R/NC</td>
<td>124.00</td>
<td>125.00</td>
<td>249.25</td>
</tr>
<tr>
<td>AZ</td>
<td>86011-5717</td>
<td><a href="mailto:alin338@anc.com">alin338@anc.com</a></td>
<td>06/31/1956</td>
<td>CA</td>
<td>101.00</td>
<td>18.89</td>
<td>119.89</td>
</tr>
<tr>
<td>AZ</td>
<td>88044-0098</td>
<td><a href="mailto:carol@cmmail.net">carol@cmmail.net</a></td>
<td>02/14/1954</td>
<td>MA</td>
<td>113.00</td>
<td>113.00</td>
<td>113.34</td>
</tr>
<tr>
<td>AZ</td>
<td>85705-4453</td>
<td><a href="mailto:ertmerle@pm.com">ertmerle@pm.com</a></td>
<td>04/15/1954</td>
<td>IA</td>
<td>163.00</td>
<td>163.00</td>
<td>326.43</td>
</tr>
<tr>
<td>CA</td>
<td>91103-4627</td>
<td><a href="mailto:aragon@pced.net">aragon@pced.net</a></td>
<td>08/12/1951</td>
<td>CA/MA</td>
<td>103.00</td>
<td>103.00</td>
<td>39.72</td>
</tr>
<tr>
<td>CA</td>
<td>92407-4657</td>
<td><a href="mailto:mmonnonoo3@mount.com">mmonnonoo3@mount.com</a></td>
<td>10/13/1943</td>
<td>CA/IN</td>
<td>120.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>CA</td>
<td>92407-2038</td>
<td><a href="mailto:hrnudl@statek.com">hrnudl@statek.com</a></td>
<td>04/25/1943</td>
<td>CA/R/IN</td>
<td>126.00</td>
<td>126.00</td>
<td>252.73</td>
</tr>
<tr>
<td>CA</td>
<td>91125-4958</td>
<td><a href="mailto:severs@cco.com">severs@cco.com</a></td>
<td>04/24/1947</td>
<td>CA/R/IN</td>
<td>119.00</td>
<td>119.00</td>
<td>119.74</td>
</tr>
</tbody>
</table>

11-3
Setting OnDataChange Events

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

  ```plaintext
  ' 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:

<table>
<thead>
<tr>
<th>Target Field Name</th>
<th>Target Field Expression</th>
<th>Description</th>
<th>Type</th>
<th>Size</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 State</td>
<td>=varState</td>
<td></td>
<td>Text</td>
<td>15</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>2 Number_of_Accounts</td>
<td>=varCounter</td>
<td></td>
<td>Text</td>
<td>15</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>3 Total_Balance_of_Accounts</td>
<td>=varBalance</td>
<td></td>
<td>Text</td>
<td>15</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Record No</th>
<th>State</th>
<th>Number_of_Accounts</th>
<th>Total_Balance_of_Accounts</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AK</td>
<td>2</td>
<td>322.97</td>
</tr>
<tr>
<td>2</td>
<td>AL</td>
<td>1</td>
<td>170.85</td>
</tr>
<tr>
<td>3</td>
<td>AR</td>
<td>1</td>
<td>527.01</td>
</tr>
<tr>
<td>4</td>
<td>AZ</td>
<td>6</td>
<td>1631.4</td>
</tr>
<tr>
<td>5</td>
<td>CA</td>
<td>25</td>
<td>7636.72</td>
</tr>
<tr>
<td>6</td>
<td>CO</td>
<td>3</td>
<td>920.32</td>
</tr>
<tr>
<td>7</td>
<td>CT</td>
<td>2</td>
<td>754.84</td>
</tr>
<tr>
<td>8</td>
<td>DC</td>
<td>2</td>
<td>430.07</td>
</tr>
<tr>
<td>9</td>
<td>DE</td>
<td>1</td>
<td>57.07</td>
</tr>
<tr>
<td>10</td>
<td>FL</td>
<td>10</td>
<td>3269.58</td>
</tr>
<tr>
<td>11</td>
<td>GA</td>
<td>7</td>
<td>1560.43</td>
</tr>
<tr>
<td>12</td>
<td>HI</td>
<td>2</td>
<td>46.32</td>
</tr>
<tr>
<td>13</td>
<td>IA</td>
<td>2</td>
<td>1004.04</td>
</tr>
<tr>
<td>14</td>
<td>ID</td>
<td>1</td>
<td>760.5</td>
</tr>
<tr>
<td>15</td>
<td>IL</td>
<td>10</td>
<td>4933.69</td>
</tr>
</tbody>
</table>

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.

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

<table>
<thead>
<tr>
<th>State</th>
<th>Zip</th>
<th>Email</th>
<th>Birth Date</th>
<th>Favorites</th>
<th>Standard I Payments</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>OH</td>
<td>44060-1300</td>
<td><a href="mailto:evan@esik.com">evan@esik.com</a></td>
<td>02/28/1979</td>
<td>XA</td>
<td>101.00</td>
<td>100.00</td>
</tr>
<tr>
<td>IL</td>
<td>60537-4596</td>
<td><a href="mailto:mapper@cert.net">mapper@cert.net</a></td>
<td>06/30/1975</td>
<td>BA</td>
<td>144.00</td>
<td>144.00</td>
</tr>
<tr>
<td>OH</td>
<td>44355-4002</td>
<td><a href="mailto:dhoni@esikon.net">dhoni@esikon.net</a></td>
<td>11/02/1981</td>
<td>EBB</td>
<td>126.00</td>
<td>126.00</td>
</tr>
<tr>
<td>MA</td>
<td>2453</td>
<td><a href="mailto:hazek@benfors.net">hazek@benfors.net</a></td>
<td>12/15/1982</td>
<td>JARIB</td>
<td>127.00</td>
<td>127.00</td>
</tr>
<tr>
<td>LA</td>
<td>70303-4518</td>
<td><a href="mailto:din333@gmail.com">din333@gmail.com</a></td>
<td>12/22/1985</td>
<td>EOEAFIKIA</td>
<td>142.00</td>
<td>142.00</td>
</tr>
<tr>
<td>SC</td>
<td>29303-9388</td>
<td><a href="mailto:cosidas@gi.com">cosidas@gi.com</a></td>
<td>02/21/1989</td>
<td>SS</td>
<td>151.00</td>
<td>151.00</td>
</tr>
<tr>
<td>CA</td>
<td>10350-1177</td>
<td><a href="mailto:deboris@r3ell.com">deboris@r3ell.com</a></td>
<td>07/31/1960</td>
<td>MAE03SB</td>
<td>113.00</td>
<td>113.00</td>
</tr>
<tr>
<td>NY</td>
<td>12561-0223</td>
<td><a href="mailto:dianne@matr.com">dianne@matr.com</a></td>
<td>06/18/1980</td>
<td>EC</td>
<td>147.00</td>
<td>147.00</td>
</tr>
<tr>
<td>MD</td>
<td>20550-2300</td>
<td>02/23/1970</td>
<td>FA</td>
<td>103.00</td>
<td>103.00</td>
<td>103.00</td>
</tr>
<tr>
<td>CA</td>
<td>91106</td>
<td><a href="mailto:aragon@papco.net">aragon@papco.net</a></td>
<td>08/12/1981</td>
<td>RANA</td>
<td>103.00</td>
<td>103.00</td>
</tr>
<tr>
<td>WA</td>
<td>54702-4608</td>
<td><a href="mailto:my2@xav.com">my2@xav.com</a></td>
<td>02/22/1982</td>
<td>FB</td>
<td>103.00</td>
<td>103.00</td>
</tr>
<tr>
<td>IN</td>
<td>47359-1435</td>
<td><a href="mailto:my2@xanr.com">my2@xanr.com</a></td>
<td>02/28/1974</td>
<td>CS</td>
<td>123.00</td>
<td>123.00</td>
</tr>
<tr>
<td>CO</td>
<td>01506-3483</td>
<td><a href="mailto:castlewood@meathlink.net">castlewood@meathlink.net</a></td>
<td>12/07/1969</td>
<td>AC</td>
<td>107.00</td>
<td>107.00</td>
</tr>
<tr>
<td>WA</td>
<td>23301-3455</td>
<td><a href="mailto:pjohnson@meathlink.net">pjohnson@meathlink.net</a></td>
<td>11/02/1969</td>
<td>AC</td>
<td>107.00</td>
<td>107.00</td>
</tr>
<tr>
<td>IL</td>
<td>61252</td>
<td><a href="mailto:mavin@sparkland.net">mavin@sparkland.net</a></td>
<td>12/15/1968</td>
<td>SA</td>
<td>171.00</td>
<td>171.00</td>
</tr>
<tr>
<td>FL</td>
<td>32608-2349</td>
<td><a href="mailto:manome@hotmail.com">manome@hotmail.com</a></td>
<td>09/26/1971</td>
<td>GG</td>
<td>126.00</td>
<td>126.00</td>
</tr>
<tr>
<td>IL</td>
<td>60537-8069</td>
<td><a href="mailto:chung@onmath.com">chung@onmath.com</a></td>
<td>05/05/1954</td>
<td>LA</td>
<td>160.00</td>
<td>160.00</td>
</tr>
<tr>
<td>TX</td>
<td>78213-3318</td>
<td><a href="mailto:ibbomh@hotmaill.com">ibbomh@hotmaill.com</a></td>
<td>09/08/1978</td>
<td>XANTA</td>
<td>127.00</td>
<td>127.00</td>
</tr>
<tr>
<td>NC</td>
<td>28216-2576</td>
<td><a href="mailto:jake.giese@jcsu.net">jake.giese@jcsu.net</a></td>
<td>10/04/1968</td>
<td>ED</td>
<td>140.00</td>
<td>140.00</td>
</tr>
</tbody>
</table>

11-5
Setting OnDataChange Events

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

Using Buffered Put Tree to Create Hierarchical Records

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**
SampleDir\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.

12-1
Using Buffered Put Tree to Create Hierarchical Records

<table>
<thead>
<tr>
<th>Source Information</th>
<th>Connection</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Source Connector: <strong>ASCII (Delimited)</strong></td>
</tr>
<tr>
<td></td>
<td>Source File/URI: <strong>SamplesDir\Data\src_BufferedPut.asc</strong></td>
</tr>
<tr>
<td>Description</td>
<td>The source ASCII file contains 83 records of account information.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Target Information</th>
<th>Connection</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Target Connector: <strong>XML</strong></td>
</tr>
<tr>
<td></td>
<td>Output Mode: <strong>Replace File/Table</strong></td>
</tr>
<tr>
<td></td>
<td>Target File/URI: <strong>SamplesDir\Data\trag_BufferedPut.xml</strong></td>
</tr>
<tr>
<td>Description</td>
<td>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 <code>&lt;All&gt;</code> view is selected.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Procedure Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>To accomplish this transformation in Map Designer, the following was done:</td>
</tr>
</tbody>
</table>

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.

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.

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).

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.

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:
Using Buffered PUT Tree to Create Hierarchical Records

<table>
<thead>
<tr>
<th>Record No</th>
<th>Account Number</th>
<th>CustomerInfo</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10019</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>8</td>
<td>10026</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>15</td>
<td>10035</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>22</td>
<td>10041</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>29</td>
<td>10047</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>36</td>
<td>10054</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>43</td>
<td>10061</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>50</td>
<td>10067</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>57</td>
<td>10073</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>64</td>
<td>10080</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>71</td>
<td>10086</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>78</td>
<td>10092</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>85</td>
<td>10099</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>92</td>
<td>10105</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>99</td>
<td>10111</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>106</td>
<td>10117</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>113</td>
<td>10123</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>119</td>
<td>&lt;null&gt;</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Record No</th>
<th>FirstName</th>
<th>LastName</th>
<th>DependentInfo</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Bobbi</td>
<td>Andt</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>9</td>
<td>Carol</td>
<td>Braun</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>16</td>
<td>Connie</td>
<td>Caterton</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>23</td>
<td>Darlene</td>
<td>Dellenmann</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>30</td>
<td>Dick</td>
<td>Dunbar</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>37</td>
<td>Duane</td>
<td>Feavel</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>44</td>
<td>Glenn</td>
<td>Guiman</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>51</td>
<td>Jane</td>
<td>Haugner</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>59</td>
<td>Jim</td>
<td>Issacson</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>65</td>
<td>John</td>
<td>Kamp</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>72</td>
<td>Joyce</td>
<td>Keatech</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>79</td>
<td>Larry</td>
<td>Lesperance</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>86</td>
<td>Lori</td>
<td>Ludwig</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>93</td>
<td>Mary</td>
<td>Mayerle</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>100</td>
<td>Mary</td>
<td>Novak</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>107</td>
<td>Mike</td>
<td>Philips</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>114</td>
<td>Pom</td>
<td>Nicholson</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>119</td>
<td>&lt;null&gt;</td>
<td>&lt;null&gt;</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

- **DependentInfo** has many more records because it lists all the records for the dependent fields and each customer happens to have 5 dependents.
<table>
<thead>
<tr>
<th>Record No</th>
<th>FirstName</th>
<th>LastName</th>
<th>Address</th>
<th>City</th>
<th>State</th>
<th>Zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Bruce</td>
<td>Beecher</td>
<td>1037 W Wisconsin Ave</td>
<td>Smithville</td>
<td>AK</td>
<td>99140-1538</td>
</tr>
<tr>
<td>4</td>
<td>Bruce</td>
<td>Beyer</td>
<td>188 E Wisconsin Ave</td>
<td>Jones town</td>
<td>AK</td>
<td>99143-1803</td>
</tr>
<tr>
<td>5</td>
<td>Dutch</td>
<td>Dobbs</td>
<td>108 Hilltop Ct</td>
<td>Smithville</td>
<td>AK</td>
<td>99156-3200</td>
</tr>
<tr>
<td>6</td>
<td>Carla</td>
<td>Bierer</td>
<td>110 Fox River Dr</td>
<td>Smithville</td>
<td>AK</td>
<td>99156-3425</td>
</tr>
<tr>
<td>7</td>
<td>Carol</td>
<td>Brauer</td>
<td>110 W North Water St</td>
<td>Jones town</td>
<td>AK</td>
<td>99216-1144</td>
</tr>
<tr>
<td>10</td>
<td>Cherri</td>
<td>Biskup</td>
<td>1222 Milwaukee St</td>
<td>Jones town</td>
<td>AK</td>
<td>99431-4175</td>
</tr>
<tr>
<td>11</td>
<td>Chuck</td>
<td>Buss</td>
<td>1134 S Franklin St</td>
<td>Everett</td>
<td>AK</td>
<td>99401-9903</td>
</tr>
<tr>
<td>12</td>
<td>Chuck</td>
<td>Carpenter</td>
<td>115 S Dew St</td>
<td>Smithville</td>
<td>AK</td>
<td>99433-9418</td>
</tr>
<tr>
<td>13</td>
<td>Chuck</td>
<td>Carr</td>
<td>1151 Valley Fair Mall</td>
<td>Jones town</td>
<td>AK</td>
<td>99451</td>
</tr>
<tr>
<td>14</td>
<td>Colleen</td>
<td>Casperson</td>
<td>120 N Morrison St</td>
<td>Smithville</td>
<td>AK</td>
<td>99501</td>
</tr>
<tr>
<td>17</td>
<td>Connie</td>
<td>Clay</td>
<td>121 N Douglas St # R</td>
<td>Smithville</td>
<td>AK</td>
<td>99501</td>
</tr>
<tr>
<td>18</td>
<td>Craig</td>
<td>Coller</td>
<td>1216 W Wisconsin Ave</td>
<td>Nicetown</td>
<td>AK</td>
<td>99590</td>
</tr>
<tr>
<td>19</td>
<td>Dan</td>
<td>Coppinger</td>
<td>1221 W Lawe St</td>
<td>Smithville</td>
<td>AK</td>
<td>99590-2104</td>
</tr>
<tr>
<td>20</td>
<td>Don</td>
<td>Dugan</td>
<td>1222 N Superior St</td>
<td>Smithville</td>
<td>AK</td>
<td>99590-2930</td>
</tr>
<tr>
<td>21</td>
<td>Darlene</td>
<td>Dantzer</td>
<td>120 W Wisconsin Ave</td>
<td>Dime Box</td>
<td>AK</td>
<td>99801-4848</td>
</tr>
<tr>
<td>22</td>
<td>Daron</td>
<td>Dinell</td>
<td>1300 E Calumet St</td>
<td>Smithville</td>
<td>AK</td>
<td>99903-3909</td>
</tr>
<tr>
<td>25</td>
<td>David</td>
<td>Dockay</td>
<td>1302 S Rigter</td>
<td>Smithville</td>
<td>AK</td>
<td>99903-2652</td>
</tr>
<tr>
<td>26</td>
<td>Dedra</td>
<td>Dorsey</td>
<td>1320 S Lincoln St</td>
<td>Smithville</td>
<td>AK</td>
<td>99903-3065</td>
</tr>
<tr>
<td>27</td>
<td>Denise</td>
<td>Dunnington</td>
<td>1396 Ridgeway Court</td>
<td>Smithville</td>
<td>AK</td>
<td>99911</td>
</tr>
<tr>
<td>28</td>
<td>Diane</td>
<td>Dugan</td>
<td>144 N Mail Dr</td>
<td>Smithville</td>
<td>AK</td>
<td>99911</td>
</tr>
<tr>
<td>29</td>
<td>Dinah</td>
<td>Dugan</td>
<td>1489 Earl St</td>
<td>Dime Box</td>
<td>AK</td>
<td>99911</td>
</tr>
<tr>
<td>32</td>
<td>Dominic</td>
<td>Earle</td>
<td>150 W Green Bay Pk</td>
<td>Lonesome</td>
<td>AK</td>
<td>99911</td>
</tr>
<tr>
<td>33</td>
<td>Don</td>
<td>Elly</td>
<td>1620 S Lawe St</td>
<td>Smithville</td>
<td>AK</td>
<td>99911</td>
</tr>
<tr>
<td>34</td>
<td>Donna</td>
<td>Erdmann</td>
<td>1737 W Pied Dr</td>
<td>Smithville</td>
<td>AK</td>
<td>99911</td>
</tr>
<tr>
<td>35</td>
<td>Donna</td>
<td>Esser</td>
<td>1831 N Richmond St #</td>
<td>Jones town</td>
<td>AK</td>
<td>99911</td>
</tr>
<tr>
<td>38</td>
<td>Emmett</td>
<td>Enos</td>
<td>1818 N Maple St</td>
<td>Smithville</td>
<td>AK</td>
<td>99911-2781</td>
</tr>
</tbody>
</table>

12-5
Using Buffered Put Tree to Create Hierarchical Records

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

<table>
<thead>
<tr>
<th>Record No</th>
<th>Record Name</th>
<th>Field Name</th>
<th>Field Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AccountInfo</td>
<td>AccountNumber</td>
<td>10019</td>
</tr>
<tr>
<td></td>
<td>CustomerInfo</td>
<td>&lt;null&gt;</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>CustomerInfo</td>
<td>FirstName</td>
<td>Bobbi</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LastName</td>
<td>Andi</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DependentInfo</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>3</td>
<td>DependentInfo</td>
<td>FirstName</td>
<td>Bruce</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LastName</td>
<td>Beecher</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Address</td>
<td>1037 W Wisconsin Ave</td>
</tr>
<tr>
<td></td>
<td></td>
<td>City</td>
<td>Smithville</td>
</tr>
<tr>
<td></td>
<td></td>
<td>State</td>
<td>AK</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Zip</td>
<td>95140-1539</td>
</tr>
<tr>
<td>4</td>
<td>DependentInfo</td>
<td>FirstName</td>
<td>Bruce</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LastName</td>
<td>Beyer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Address</td>
<td>108 E Wisconsin Ave</td>
</tr>
<tr>
<td></td>
<td></td>
<td>City</td>
<td>Jonestown</td>
</tr>
<tr>
<td></td>
<td></td>
<td>State</td>
<td>AK</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Zip</td>
<td>95140-1539</td>
</tr>
</tbody>
</table>

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


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

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)
Aggregating Records

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.

![Transformation and Map Properties](image)

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:

```plaintext
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 **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_Total")
   - **C_Pymnt_Min**
Aggregating Records

If
Targets(0).Records("City_Payment").Fields("C_Pymnt_Min") = 0 Then
Fields("Payment")
ElseIf
Fields("Payment")<Targets(0).Records("City_Payment").Fields("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_Count") + 1

- **St_Pymnt_Total**
  Fields("Payment") + 
  Targets(0).Records("State_Payment").Fields("St_Pymnt_Total")

- **St_Pymnt_Min**
  If
  Targets(0).Records("State_Payment").Fields("St_Pymnt_Min") = 0 Then
  Fields("Payment")
  ElseIf
  Fields("Payment") >
  Targets(0).Records("State_Payment").Fields("St_Pymnt_Min")
End If
Then
Targets(0).Records("State_Payment").Fields("St_Pymnt_Min")
Else
Fields("Payment")
End If

• **St_Pymnt_Max**

If
Targets(0).Records("State_Payment").Fields("St_Pymnt_Max") = 0 Then
Fields("Payment")
ElseIf Fields("Payment") >
Targets(0).Records("State_Payment").Fields("St_Pymnt_Max") Then
Fields("Payment")
Else
Targets(0).Records("State_Payment").Fields("St_Pymnt_Max")
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

13-7
Aggregating Records

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**

Manipulating Binary Dates at the Bit Level

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.
Manipulating Binary Dates at the Bit Level

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.

<table>
<thead>
<tr>
<th>Record No</th>
<th>Field1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5000</td>
</tr>
<tr>
<td>2</td>
<td>550</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
</tr>
<tr>
<td>4</td>
<td>5150</td>
</tr>
<tr>
<td>5</td>
<td>200</td>
</tr>
<tr>
<td>6</td>
<td>250</td>
</tr>
<tr>
<td>7</td>
<td>300</td>
</tr>
<tr>
<td>8</td>
<td>350</td>
</tr>
<tr>
<td>9</td>
<td>400</td>
</tr>
<tr>
<td>10</td>
<td>450</td>
</tr>
<tr>
<td>11</td>
<td>500</td>
</tr>
<tr>
<td>12</td>
<td>550</td>
</tr>
<tr>
<td>13</td>
<td>600</td>
</tr>
<tr>
<td>14</td>
<td>650</td>
</tr>
<tr>
<td>15</td>
<td>700</td>
</tr>
<tr>
<td>16</td>
<td>750</td>
</tr>
<tr>
<td>17</td>
<td>800</td>
</tr>
<tr>
<td>18</td>
<td>850</td>
</tr>
<tr>
<td>19</td>
<td>900</td>
</tr>
<tr>
<td>20</td>
<td>950</td>
</tr>
<tr>
<td>21</td>
<td>1000</td>
</tr>
</tbody>
</table>

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. This 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

<table>
<thead>
<tr>
<th>Column #</th>
<th>Column Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Column 1</td>
<td>SOURCE</td>
<td>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.</td>
</tr>
<tr>
<td>Column 2</td>
<td>DATE</td>
<td>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 yyyyymmdd 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.</td>
</tr>
<tr>
<td>Column 3</td>
<td>YEAR</td>
<td>Here we converted the “year bits” in the source to a single target character field for comparison to the data in column 2.</td>
</tr>
<tr>
<td>Column 4</td>
<td>MON</td>
<td>Here we converted the “month bits” in the source to a single target character field for comparison to the data in column 2.</td>
</tr>
<tr>
<td>Column 5</td>
<td>DAY</td>
<td>Here we converted the “day bits” in the source to a single target character field for comparison to the data in column 2.</td>
</tr>
</tbody>
</table>
Manipulating Binary Dates at the Bit Level

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**:
   ```vba
   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.
     ```vba
     Day1 = Records("Record1").Fields("Field1") And 31
     ```
   - This expression calculates the value of the month by comparing the bit values.
     ```vba
     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.
     ```vba
     Year1 = (Records("Record1").Fields("Field1")/512) And 127
     ```
     ```vba
     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**:
   ```vba
   Year1 = (Records("Record1").Fields("Field1")/512) And 127
   ```
   ```vba
   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.
Manipulating Binary Dates at the Bit Level

\[
\begin{array}{ccccccccc}
0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
64 & 32 & 16 & 8 & 4 & 2 & 1 & 16 & 8 \\
\end{array}
\]

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.
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
Complex Date Filtering

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.

<table>
<thead>
<tr>
<th>Record No</th>
<th>Record Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>341</td>
<td>12/07/1996</td>
</tr>
<tr>
<td>342</td>
<td>12/08/1996</td>
</tr>
<tr>
<td>343</td>
<td>12/09/1996</td>
</tr>
<tr>
<td>344</td>
<td>12/10/1996</td>
</tr>
<tr>
<td>345</td>
<td>12/11/1996</td>
</tr>
<tr>
<td>346</td>
<td>12/12/1996</td>
</tr>
<tr>
<td>347</td>
<td>12/13/1996</td>
</tr>
<tr>
<td>348</td>
<td>12/14/1996</td>
</tr>
<tr>
<td>349</td>
<td>12/15/1996</td>
</tr>
<tr>
<td>350</td>
<td>12/16/1996</td>
</tr>
<tr>
<td>351</td>
<td>12/17/1996</td>
</tr>
<tr>
<td>352</td>
<td>12/18/1996</td>
</tr>
<tr>
<td>353</td>
<td>12/19/1996</td>
</tr>
<tr>
<td>354</td>
<td>12/20/1996</td>
</tr>
<tr>
<td>355</td>
<td>12/21/1996</td>
</tr>
<tr>
<td>356</td>
<td>12/22/1996</td>
</tr>
<tr>
<td>357</td>
<td>12/23/1996</td>
</tr>
<tr>
<td>358</td>
<td>12/24/1996</td>
</tr>
<tr>
<td>359</td>
<td>12/25/1996</td>
</tr>
<tr>
<td>360</td>
<td>12/26/1996</td>
</tr>
<tr>
<td>361</td>
<td>12/27/1996</td>
</tr>
<tr>
<td>362</td>
<td>12/28/1996</td>
</tr>
<tr>
<td>363</td>
<td>12/29/1996</td>
</tr>
<tr>
<td>364</td>
<td>12/30/1996</td>
</tr>
<tr>
<td>365</td>
<td>12/31/1996</td>
</tr>
</tbody>
</table>

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:

\[ \text{DatePart}("m", \text{DateValMask}\left(\text{Fields}("Date"), "mm/dd/yyyy"\right)) \geq 6 \text{ And } \text{DatePart}("m", \text{DateValMask}\left(\text{Fields}("Date"), "mm/dd/yyyy"\right)) \leq 8 \]

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

\[ \text{DatePart}("d", \text{DateValMask}\left(\text{Fields}("Date"), "mm/dd/yyyy"\right)) \geq 1 \text{ And } \text{DatePart}("d", \text{DateValMask}\left(\text{Fields}("Date"), "mm/dd/yyyy"\right)) \leq 31 \]

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

\[ \text{DatePart}("yyyy", \text{DateValMask}\left(\text{Fields}("Date"), "mm/dd/yyyy"\right)) = 1998 \]

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.
Complex Date Filtering
Working with DJRowSet and Arrays

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**

Source Connector: **ASCII (Delimited)**
Working with DJRowSet and Arrays

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  
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.

<table>
<thead>
<tr>
<th>Record Name</th>
<th>Len</th>
<th>Lock</th>
<th>Schema Origin</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>94</td>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>94</td>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>95</td>
<td>No</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

a. Append 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 **OnDataChange** 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](image.png) 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:

      ```plaintext
      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
Working with DJRowSet and Arrays

- **Item Information**:
  Category
  ProductNumber
  Quantity
  UnitCost
  Total
  ShipmentMethodCode

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

- **Account Information**:
  PurchaseOrderInformation

- **Purchase Order Information**
  Item Information

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

<table>
<thead>
<tr>
<th>Target Field Name</th>
<th>Target Record</th>
<th>Target Field Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>ItemNumber</td>
<td>Item Information</td>
<td>=cntr</td>
</tr>
<tr>
<td>Number_of_Items</td>
<td>PurchaseOrderInformation</td>
<td>=varItemNumber</td>
</tr>
<tr>
<td>POTotal</td>
<td>PurchaseOrderInformation</td>
<td>=varPOTotal</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Record No</th>
<th>POID</th>
<th>Date</th>
<th>Number of Items</th>
<th>POTotal</th>
<th>Item Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>2002-12-23</td>
<td>6</td>
<td>17,079,000,000,000,000</td>
<td>null</td>
</tr>
<tr>
<td>5</td>
<td>02-336361</td>
<td>2002-12-17</td>
<td>2</td>
<td>867,000,000,000,000,000</td>
<td>null</td>
</tr>
<tr>
<td>9</td>
<td>02-336362</td>
<td>2002-12-17</td>
<td>2</td>
<td>50,940,000,000,000,000</td>
<td>null</td>
</tr>
<tr>
<td>13</td>
<td>01-674005</td>
<td>2002-12-23</td>
<td>1</td>
<td>41,039,000,000,000,000</td>
<td>null</td>
</tr>
<tr>
<td>15</td>
<td>01-674005</td>
<td>2002-12-24</td>
<td>2</td>
<td>299,059,000,000,000,000</td>
<td>null</td>
</tr>
<tr>
<td>20</td>
<td>01-438024</td>
<td>2002-12-17</td>
<td>2</td>
<td>71,088,000,000,000,000</td>
<td>null</td>
</tr>
<tr>
<td>24</td>
<td>01-438024</td>
<td>2002-12-18</td>
<td>4</td>
<td>152,000,000,000,000,000</td>
<td>null</td>
</tr>
<tr>
<td>30</td>
<td>01-950009</td>
<td>2002-12-08</td>
<td>1</td>
<td>43,289,000,000,000,000</td>
<td>null</td>
</tr>
<tr>
<td>33</td>
<td>02-109614</td>
<td>2002-12-10</td>
<td>6</td>
<td>933,718,000,000,000,000</td>
<td>null</td>
</tr>
<tr>
<td>41</td>
<td>02-109616</td>
<td>2002-12-12</td>
<td>4</td>
<td>131,888,000,000,000,000</td>
<td>null</td>
</tr>
<tr>
<td>47</td>
<td>02-792777</td>
<td>2002-12-26</td>
<td>3</td>
<td>169,928,000,000,000,000</td>
<td>null</td>
</tr>
<tr>
<td>50</td>
<td>&lt;null&gt;</td>
<td></td>
<td>8</td>
<td>447,628,888,888,888,888</td>
<td>null</td>
</tr>
</tbody>
</table>

**ItemInformation** 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.
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.

Dynamic SQL Lookup

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
Dynamic SQL Lookup

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

Procedure
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.

---

Dynamic SQL Lookup
Dynamic SQL Lookup with Error Handling

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
Dynamic SQL Lookup with Error Handling

**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:

   ```vbs
   ' 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:

```vba
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:

```vba
' 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**


Search for the words “transformation event handlers” in the online documentation.
Dynamic SQL Lookup with Error Handling
Dynamic SQL Lookup with Reject Records Handling

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
Dynamic SQL Lookup with Reject Records Handling

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_dsdlreject.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 Transformation, 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.

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

8 On the Map tab, in the Source tree, we set an After Every Record 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 On Error event handler.
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.

<table>
<thead>
<tr>
<th>Action Name</th>
<th>Action Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ClearMapPut Record</td>
</tr>
<tr>
<td>2</td>
<td>Resume</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Required</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>target name</td>
<td>Yes</td>
<td>Reject</td>
</tr>
<tr>
<td>record layout</td>
<td>Yes</td>
<td>R1</td>
</tr>
<tr>
<td>count</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>counter variable</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>buffered</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

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")
```
Dynamic SQL Lookup with Reject Records Handling

**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**


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

Using EDI X12 Iterator to Read Messages

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
Using EDI X12 Iterator to Read Messages

(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):

\[
A = \text{myMsg.Properties("DJFT PieceNumber")}
\]
\[
B = \text{myMsg.Properties("DJFT TotalPieces")}
\]
\[
C = \text{myMsg.Properties("DJFT Priority")}
\]

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

\[
CR = \text{Chr}(13)
\]

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:

\[
\]

If \( C = 1 \) Then
\[
\text{MsgBox("Last Message Received")}
\]
End If

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

![Message Box Example]

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

\[
\text{Project("Get\_Next\_ISA").ReturnCode = 0}
\]

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:
Using EDI X12 Iterator to Read Messages

**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**
Using EDI X12 Iterator to Read Messages
CONNECTOR-SPECIFIC SAMPLES
Microsoft Dynamics GP 10: Updating 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
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:

<table>
<thead>
<tr>
<th>Action</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClearMap</td>
<td>{Target},{Customer1},{},{}</td>
</tr>
<tr>
<td>Update Record</td>
<td>{Target},{Customer1},{Customer},{},{}</td>
</tr>
<tr>
<td>ClearMap</td>
<td>{Target},{Customer}Addresses1,{},{}</td>
</tr>
<tr>
<td>Update Record</td>
<td>{Target},{Customer}Addresses1,{{Customer}Addresses},{},{}</td>
</tr>
<tr>
<td>ClearMap</td>
<td>{Target},{Customer}Addresses2,{},{}</td>
</tr>
<tr>
<td>Update Record</td>
<td>{Target},{Customer}Addresses2,{{Customer}Addresses},{},{}</td>
</tr>
</tbody>
</table>
Next, we cleared the map for the Customer1 record layout.

Then we specified an Update Record action for the Customer1 record layout and Customer record.

We cleared the map for the Customer Addresses1 record layout.

We used an Update Record action to update the Customer Addresses1 record layout and the Customer record.

We repeated steps 5 and 6 to update the Addresses2 record.

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 GP 10: Updating Records
Microsoft Dynamics CRM 4.0: Inserting Records

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.xm1

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, UserID, 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 ClearMapInsertRecord 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 UpdateRecord 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

Deleting a Child Record From a Child Entity

Objectives
Removes a Account_BContact 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.cronemandemand.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 retrieve 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

## 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
Netsuite 2.6: Entering Sales Orders

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”.

25-2
In a Source record event handler, we created an AfterEveryRecord event and added the following events:

<table>
<thead>
<tr>
<th>Record Name</th>
<th>Event Name</th>
<th>Action</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Items</td>
<td>AfterEveryRecord</td>
<td>ClearMapInsert</td>
<td>{Target} {SOItems}{SalesOrder Item}[]</td>
</tr>
<tr>
<td>Opportunity</td>
<td>AfterEveryRecord</td>
<td>ClearMapInsert</td>
<td>{Target} {SalesOrder}{SalesOrder}[]</td>
</tr>
</tbody>
</table>

**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 external ID 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”.

25-3
<table>
<thead>
<tr>
<th>Results</th>
<th>Run the map. Note that sales order data from the Opportunity fields in the XML source are inserted into the NetSuite target fields.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference</td>
<td>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.</td>
</tr>
</tbody>
</table>
Netsuite 2.6: Adding Addresses to Addressbook

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
Netsuite 2.6: Adding Addresses to Addressbook

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 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 \textbf{AfterEveryRecord} event and add the following events:

<table>
<thead>
<tr>
<th>Record Name</th>
<th>Event Name</th>
<th>Action</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>SFDC</td>
<td>AfterEveryRecord</td>
<td>ClearMapInsert</td>
<td>{Target}{Customer}{Customer}{}</td>
</tr>
<tr>
<td>SFDC</td>
<td>AfterEveryRecord</td>
<td>ClearMapInsert</td>
<td>{Target}{Address}{Customer Addressbook}{2}{cnt}</td>
</tr>
</tbody>
</table>

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 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:
\[
\text{Sources}(0)\text{Fields}(8+(5*(\text{cnt}-1)))
\]

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.

\textbf{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

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:
Conversion an EDI Source Containing an N1 Loop

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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Location</th>
<th>Option Selected</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDI_N1_Loop.edi</td>
<td>Source Connection tab</td>
<td>Source file</td>
<td>EDI (XM12) source</td>
</tr>
<tr>
<td>EDI_N1_Loop_x12.401 0.108.0.0.ds.xml</td>
<td>EDI (X12) Properties</td>
<td>SchemaFile</td>
<td>Document schema file to provide structure for source file.</td>
</tr>
<tr>
<td>EDI_N1_Loop.txt</td>
<td>Target Connection tab</td>
<td>Target file</td>
<td>Header property is set to True. This target file is created after the transformation is run.</td>
</tr>
<tr>
<td>N_Count CMPCnt</td>
<td>Transformation and Map Properties</td>
<td>Global Variables</td>
<td>Two global variables called N_Count and CMPCnt with initial values of 0.</td>
</tr>
</tbody>
</table>

**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.N1 record 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.

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.

Then we expanded the SE record and SE Event Handlers, 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.

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.

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.
Converting an EDI Source Containing an N1 Loop

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

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
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

Index 1
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
Iterators
  EDI X12 Functional Group 20-1
  File Transfer 20-5

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

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

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

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

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

2 Index
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

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

variables, global 8-1

XML target 12-2, 16-2