BLUEPHOENIX

ASNA

Migrating Applications User's Guide
For ASNA Monarch® 4.0

Migrating Applications User's Guide
For ASNA Monarch® 4.0

Information in this document is subject to change without notice. Names and data used in examples are fictitious unless otherwise noted.

No component of this manual may be reproduced, disassembled, transmitted, transcribed, stored in a retrieval system, or translated into any language in any form without the written permission of ASNA (Amalgamated Software of North America).

Copyright © 2004 - 2008 ASNA - Amalgamated Software of North America. All rights reserved.

Release 4.0.2 July 2008

Contents

Chapter 1 - Introduction 1

1. Object Inspection 2
2. Visual Analysis and Strategy Generation 3
3. Source and Data Migration 5
4. Unsupported Features Resolution 5

Epilogue..6

Chapter 2 - Preparing Galleries (Object Gallery) 7

Setting Preferences 7

General Preference Directives 8
Copybook Preference Directives 9
Message File Preference Directives 9
Display File Preference Directives 9
.Net Print File Preference Directives 10
Database File Preference Directives 12

RPG Agent Migration Directives 13

Creating a Gallery 14

Gallery Objects 16
Adding Libraries 17

Dealing with the Source 17

Database Connections 17
Source Mappings Table 18
Source Search Algorithm..19
Source Search Sequence..19
*LIBL...20
Checking the Source Mappings...21
Locating Object Sources...21
Changing Source Location 22
Missing Objects..23
Synchronizing Single-Module Programs 24

Discovering Artifacts..24

Copybooks 24
Printer O-Specs..25
Data Structures 25

Chapter 3 - Working with Exhibits, Sections, and Clusters 29

Partitioning an Application 29

Exhibits....29
Sections and Clusters....29
Cluster Definitions....30
Creating Exhibit Sections....31
Create a New Exhibit....31
Defining a Section....32
Section Rules....32
Exhibit, Section, and Cluster Tabs....33
Graph Coloring....34
Exhibit Directives....35
Exhibit Call Graph....35
Exhibit Details (Statistics)....36
Exhibit DB Files Graph....36
Exhibit Magnitude....37
Section Directives....37
Section Call Graph....38
Section Details....38
Cluster Directives....39
Cluster Details....39
Cluster Call Graph....39
Working with Sections and Clusters....40
Re-Sectioning Exhibits....40
Copying Exhibits....41
Creating Group Clusters....41
Splitting Clusters....42
Additional Functions....42
Examples....43
Chapter 4 - Working with GamePlans....47
Project Types....47
Interactive....47
Non-Interactive....49
Creating GamePlans....51
Including Additional Objects....53
Establish GamePlan Directives....54
Visual Studio Options....55
Data File....56
Display File Function Keys....56
New .NET Printfiles....56

Checking the Task List 57
Using Filters 57
Summary Panel 59
Sizing Migration 60
Magnitude Density 60
Effort Density 63
Chapter 5 - Pulling the (Migration) Trigger 65
Establishing Templates and Cascading Style Sheets 65
Cascading Style Sheets 67
Display File Templates 67
Master Pages Considerations 68
Templates\WebRoot Considerations 68
Solution Builder 69
Templates and Cascading Style Sheet Considerations 70
Other Considerations 70
Chapter 6 - Migrating Display Files 71
Display Files Overview 71
Display File Directives 72
Templates 72
Template.aspx 72
Template.aspx.vr 72
Example 73
Template.aspx Code 73
Template.aspx.vr Code 73
Chapter 7 - Migrating CL Programs 75
Resources 76
Chapter 8 - Migrating RPG Programs 77
Program Migration Directives 77
Op-Code Support 78
Loading Program References 79
Generation of Migrated Code - Behind the Scenes 79
/Error AVR compiler directive 79
Migrated Visual RPG File's Header 79
New RPG Program's Wrapper Class 80
ASNA.Monarch.Program Base Class 81
ASNA.Monarch.Job class 81
Monarch-Generated Properties 82
Local Data Area 83

System i Data Area access 84
Free Format C-Specs 85
Status Data Structures........ 86
Program Status Data Structures........ 86
File Information Data Structure........ 88
Cycle Support 89
Overview........ 89
AVR Cycle 90
File Designation Keyword 92
*StartCycle 92
First Page........ 92
Matching Records 93
Level Breaks 93
*TotalCalc 94
*DetailCalc 94
Last Record 95
Cycle Printing 95
Overflow 95
Lookahead Fields........ 96
Examples 96
Embedded SQL Migration 100
Overview........ 100
Examples 103
Monarch Framework Support 109
Array Translation 109
Hexadecimal Constants 110
Compile-Time Data........ 110
AVR Program Constructor 110
Generated compile-time loading subroutines 111
Other Considerations 112
Internally-Described Files 112
Chapter 9 - Advanced RPG Data Structure Migration 113
Overlapping Data Structures........ 113
Data Structures with Gaps........ 113
Giving individual field names to elements of an array 113
Masquerading a Data Structure as a Large Character Field 115
Partitioning a Data Structure Field into Sub-Fields (Date, Time) 115
Partitioning a Data Structure Field into Sub-Fields (Coded Names) . 116

Renaming fields and mapping to arrays 116
Externally-Described DS augmented by new fields 117
Migrating Data Structures defined locally in Procedures 118
Unsupported Overlapping Data Structures 122
Chapter 10 - Migrating Print Files 123
Print File Special Considerations 123
Print File Directives 124
Migrating Externally Described Printer Files 125
Migrating Printer O-Specs 125
About Migrating Printer O-Specs 126
Monarch-Generated Record and Field Names 128
RPG Agent Handling of Printer O-Specs 131
Chapter 11 - Migrating Message Files 135
Message File Directives 135
Install Directory 135
Chapter 12 - Migrating Data 137
Using Cocoon Data Migration Agents 137
Establishing Directives 138
Viewing Data Migration Results 138
Using DataGate Database Manager 139
Copying Data 140
Copying a Logical File 140
Copying a Complete Library 141
Considerations for DataGate SQL Server 142
Differences between DataGate, DataGate /400 .NET, and DataGate for SQL Server for .NET 143
Object Considerations 143
Index (Keys) Considerations 144
Data Access Considerations 145
Locking Considerations 145
Field Considerations 147
Native SQL Server field interpretation 148
Join Considerations 149
Calling Programs/Procedures Considerations 149
Chapter 13 - Resolving Unsupported Features 151
How to Migrate Goto from a Subroutine to Mainline Code 151

Chapter 14 - Committing GamePlan Status 153

Commit GamePlan Status to Gallery Dialog 153

Chapter 15 - DataGate for iSeries and ADO: a Perspective 155

How DG/400 Works 155

Client/Server *vs*. Host Based 157

Record-Level Database Access vs. Set-based Database Access 158

DataGate's Role in Monarch 160

Conclusion.. 160

Chapter 1 - Introduction

After deciding to migrate away from the System i™ OS/400®, a planning stage has to take place. This initial planning stage is more a mental activity than a technical one, and it is preliminary to employing Monarch.

There might be important documents to review at this stage and company-wide strategies to take into account.

The **main actions** to perform during this stage are:

- Review company and application documents
- Interview with application owners and end users
- Assemble the Migration team
- Choose the next application to migrate

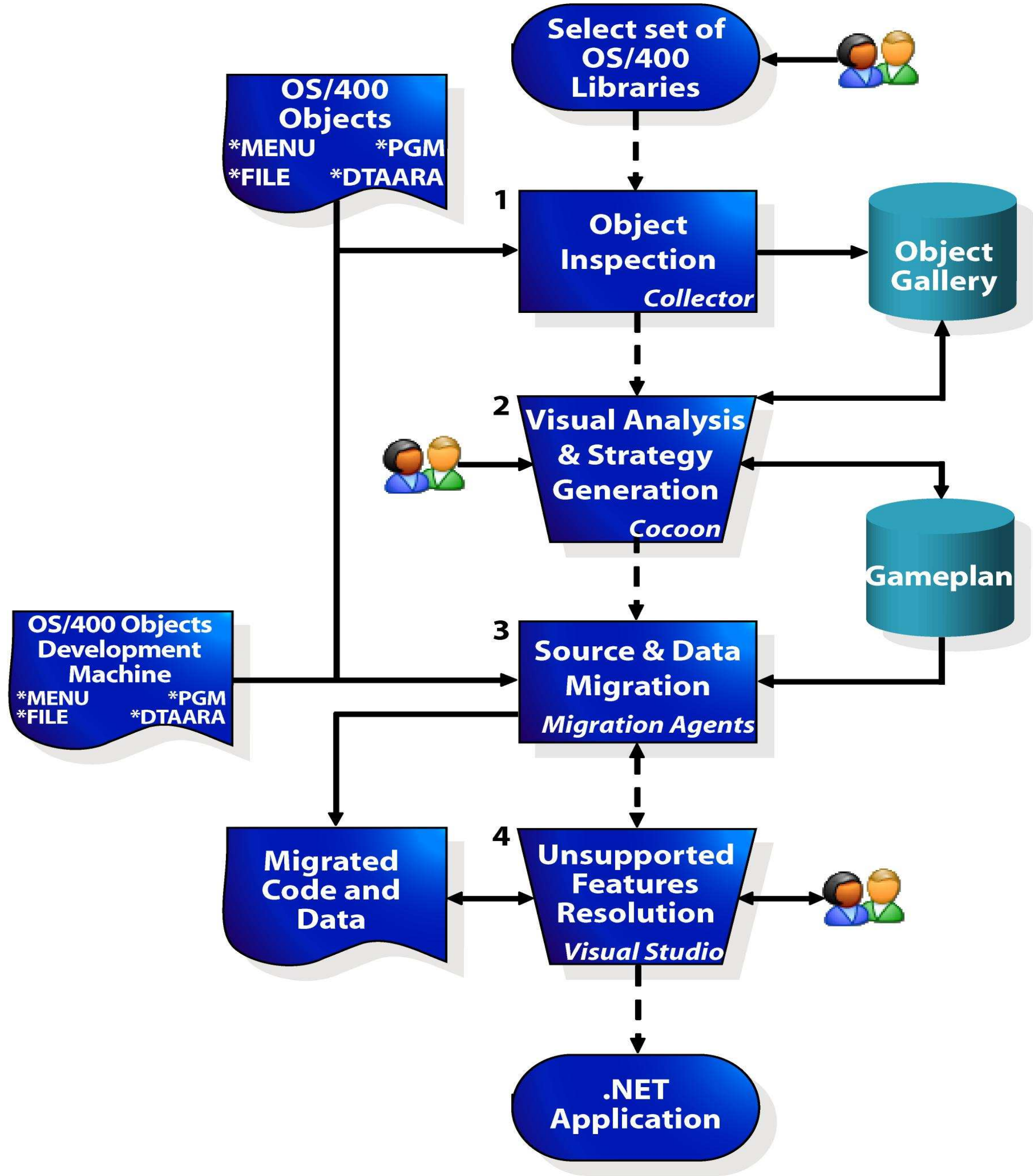
The **desired result** of this stage should yield the following:

- Identification of Applications to Migrate
- High-level description of the Applications
- Team roles and responsibilities

Critical aspects to a successful launch on the migration effort are Executive sponsorship and a Buy-in on the Migration plan from the IT department.

When the scope of the migration endeavor has been set and the organization has committed to follow through on the migration, then the Monarch Migration Process can be set in motion. This process consists of four main steps.

The following diagram depicts this process as a flow chart.



1. Object Inspection

Once the Migration team has been selected, roles have been defined, and the Application has been identified, we need to collect - in an ***Object Gallery*** - detailed information about all of the OS/400 objects involved.

The Monarch ***Cocoon*** is the workbench used to create the Object Gallery. An Object Gallery is created by selecting a database where the Gallery will reside. To populate the Gallery, a Legacy server is selected (via a database name), and the list of OS/400 libraries containing the application objects is supplied.

The Monarch ***Collector*** enumerates through the given OS/400 libraries gathering details about each of the objects found and enters them into the Gallery.

The following objects are considered when populating the Gallery (All other object types are ignored):

- Menus
- Programs and Service Programs

  Note: *Any reference to 'programs' in this document* ***also*** *refers to service programs unless otherwise noted.*

- Modules
- Database Files
- Display Files
- Printer Files
- Data Areas
- Message Files

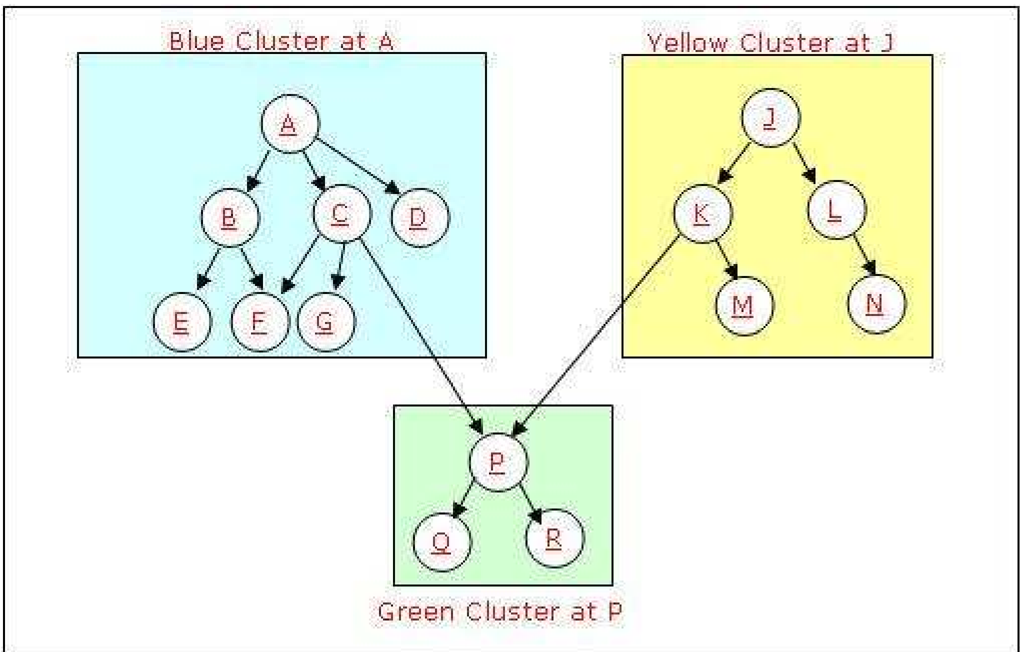
In selecting the libraries to feed the Collector, it is crucial to include the Library containing the main entry point into the application whether it is a menu or a Program. It is preferable to include all the libraries involved in the application to avoid having to repeat this step later on. However, it is possible to add more libraries after the initial collection has occurred (see Chapter 2 - Preparing Galleries).

It is imperative to run the Collector under a user profile with enough authority to all of the OS/400 objects involved in the application.

2. Visual Analysis and Strategy Generation

Important decisions must be made to determine how the different OS/400 objects will be transformed to the equivalent .NET components.

Cocoon organizes the Objects by type and provides several analysis tools to better visualize the Application as a whole, chief amongst them are **Exhibits**. An Exhibit is a graphical representation of all the programs in a Gallery grouped into one or more user defined Sections, containing Clusters. The following image shows a 15 program Gallery grouped into one Section contain three Clusters, the blue cluster starting at A, a yellow cluster at J, and the green cluster starting at P.



The ability to visualize the relationships between program clusters and to group them into sections facilities the partitioning of an application into manageable units of work (see Chapter 3).

When an application has been broken down into sections with clusters of reasonable size and function, each cluster can be turned into a GamePlan or affect its migration.

Each migrated GamePlan constitutes a Microsoft® Visual Studio® 2005 or 2008 ASNA Visual RPG® Project.

A GamePlan is defined by one or more entry points. Each entry point is a Program or Menu called from other Programs, Menus, or the command line. The GamePlan includes not only the Programs called by the entry point, but also all other objects used by the Programs involved.

Each GamePlan has associated with it a set of properties, including a Project Type, a Name, and a Location used to create a Visual Studio *Project* where the programs are crystallized.

Having good analytical skills and a deep knowledge of the application and its relationship with other applications will help in determining the best way to partition the Programs and Clusters into GamePlans.

Each Object in the GamePlan contains a **Directives** page containing pertinent information used in the next step (Program and data transformation). The collection of all of these decisions will form the aggregate representation of the Migration strategy.

To create a GamePlan, a Cluster is selected from an Exhibit, or an entry-point or top-level Program or Menu is selected from the Gallery; and a Library List is supplied. The Cocoon will crawl through the object reference web to determine all objects forming the Subsystem. This information is the basis for the initial content of the GamePlan (see Chapter 4)**.**

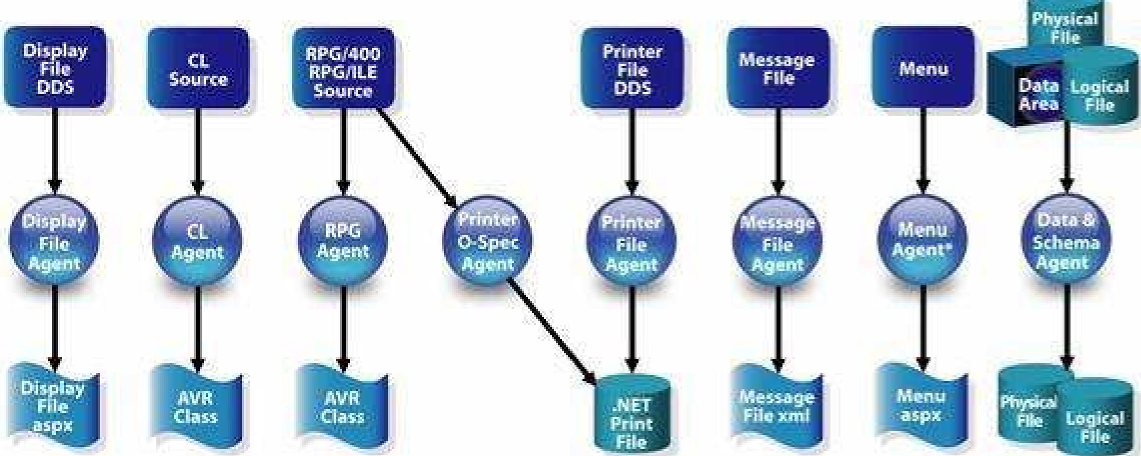
3. Source and Data Migration

Once the migration Directives are defined, the actual migration of the Application is performed by selecting Cocoon's **Migrate** menu option.

Several Migration Agents will run to convert each object in a GamePlan to its final Windows .NET form. Each Agent will use the Directives defined for each object during the previous step.

The whole process will be logged in detail with Errors separate to assist during the conflict resolution step.

There are multiple agents used to Migrate specific object types and one agent responsible for creating the Visual Studio Project and Solution. This diagram shows the agents with their inputs and outputs:



4. Unsupported Features Resolution

The Migration Agents will likely detect some problems while transforming the Legacy objects into .NET components.

The following are some ways to deal with these problems.

- Study the Migration Logs and decide how to resolve the reported problems.
- Open the generated Solution using Visual Studio and start modifying the Projects and sources by adapting the code to the .NET platform on those features that are unsupported by either AVR or .NET. Examples of these include usage of ICF files. Also, eliminate the code that is irrelevant in the new environment.
- Study any errors and warnings provided by the compiler. Use the available debugging facilities provided by the Visual Studio environment.

Epilogue

After the application has been successfully migrated from OS/400, a phase of testing (both in the lab and in the field) is necessary. Next, the application will need incorporated into the production environment of the organization and the legacy application will be retired. A new period starts at that point for the application and its developers. New enhancements, capabilities, and presentations can allow the application to serve the organization for years to come.

When the time comes to do maintenance on this application, a decision will have to be made to determine whether it is justifiable to start a new Project using a stateless/scalable application model, or to continue using the Monarch run-time model.

Chapter 2 - Preparing Galleries (Object Gallery)

Setting Preferences

One of the first steps in a migration effort is obtaining basic information about all objects involved in the applications to be migrated. The Object Inspector runs through a set of libraries on the OS/400 system to collect data for each object (i.e. Program, File, Data Area, etc.) found. All this data is collected in an object **Gallery**.

Only those object types that Monarch can deal with are added to the Gallery. Examples of objects not allowed are *JRN, *LOCALE, *OUTQ, *PNLGRP, and *USRSPC. It is possible to create multiple Galleries from a single OS/400 system, but all of the objects in a Gallery should come from a single system.

From the implementation point of view, an Object Gallery is a **Library** in the working database in which a set of database files reside containing the details of the Gallery. Indeed, the **Gallery name *is* the Library name**.

Using a DataGate Windows database (local or multi-user) in lieu of a System i, has these advantages:

- No additional libraries on the System i
- Gallery names may be 31 characters in length rather than 10

Since a Gallery is the starting point for the migration, use great care when creating and setting it up because the characteristics of the Gallery are inherited by the objects in the Gallery and when you create a **GamePlan**.

Prior to creating a Gallery, several default directive preferences must be updated to establish the basic directives for the applications objects. These preferences are used at different times - during the creation of a Gallery, locating objects, creating a GamePlan, and even on into the Migration process itself. Setting these preferences initially eliminates the need to set directives for each object independently.

These default directive preferences are set by selecting **Preferences** from the toolbar menu. There are then three preference menu options - **Directives Defaults**, **RPG Agent Directives**, and **Legacy Source Colors**.

Within Directives Defaults, there are directive settings for General preferences, Copybooks, Message Files, Display Files, Net Print Files, and Database Files.

This topic covers only the basic information but more detail can be found in the Cocoon User's Guide. The directives of note for a specific object will be included in later sections of the manual as needed. Become familiar with the preference settings available should the need to make changes occur later on.

General Preference Directives

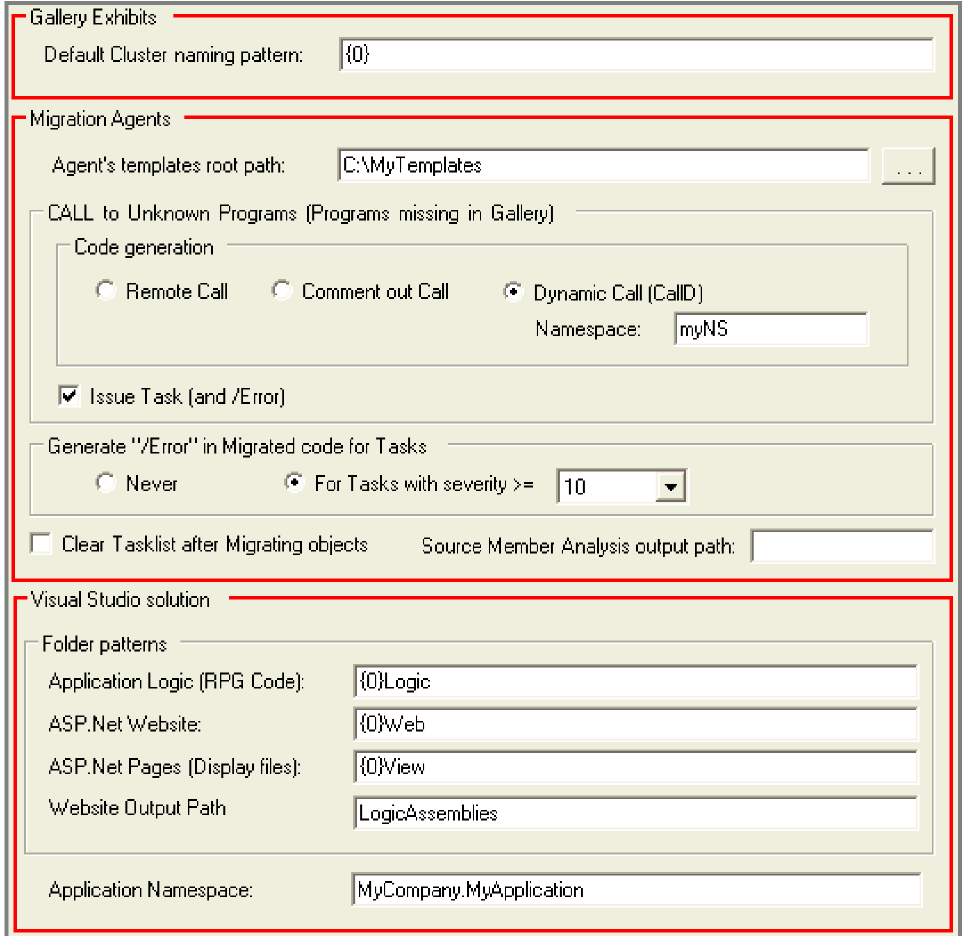
On General directives, you may want to update the Gallery Exhibits *"Default Cluster naming pattern"*, the *"Migration Agents" section,* the Visual Studio solution *"Folder patterns",* and the "*Application Namespace"* outlined below.

The *Default Cluster naming pattern* is used when Clusters are created for an Exhibit.

The *Migration Agents* section covers several areas providing direction for the migration agents. The *Agents templates root path* gives the location of your own templates and cascading style sheets covered in more detail in Chapter 5. The *Call to Unknown Programs* controls how calls are handled and if a task is to be issued. *Generate "/Error" in Migrated code for Tasks* allows you to set the level of severity for tasks that are to be reported on the task list.

In the Visual Studio solution section, the *Folder patterns* are used for naming conventions where the *BaseName* entered in the GamePlan Directive is used as the substitution variable {0} in any pattern where the placeholder is used. The *Website Output Path* is used as the subdirectory inside Bin where the common language runtime will search when loading assemblies. An assembly binding entry in Web.config indicates the search directories.

The *Application Namespace* is the default for the first level namespace and assembly directive created in the Gallery. More information on directives can be found in Chapter 4.



Copybook Preference Directives

The Copybook preference establishes the migration default location. A different location for a specific file can be set in its copybook directive. When *"Make /copy paths relative to the location of this program's source file location"* is checked, the migration agent uses the common root of the program's source path and the copybook path to issue a relative path reference for the /COPY statement. This directive can also be set for a specific program on each program directive.

Message File Preference Directives

The Message File preference establishes the migration default location. A different location for a specific file can be set in the respective message file directives.

Display File Preference Directives

These directives control function key labels, screen size, and grid settings for DDS Display Files.

Discarding labels in rows is used to throw away any redundant labels in old screens used to display the function key command description. The migrated aspx will show buttons for active function keys with nice labels (and tool tips), making the old labels redundant.

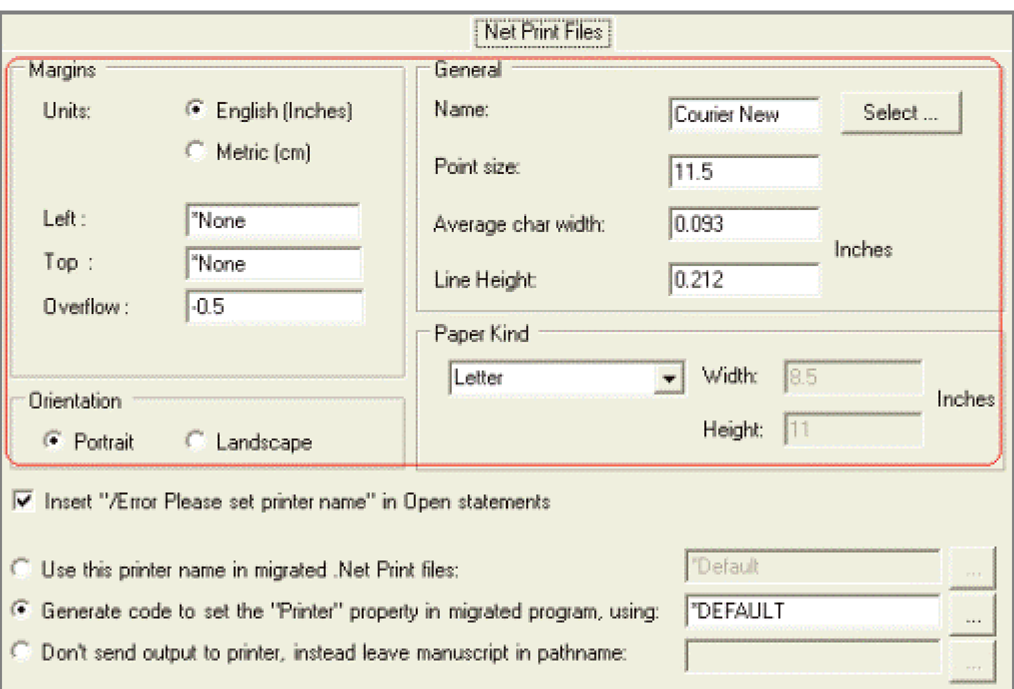
When Screen Size indicators used indicates the screen size for display files where *DS3 indicates 24x80 while *DS4 indicates 27x127.

Grid setting controls horizontal and vertical spacing. 8 horizontal and 24 vertical works well for Monarch to produce a grid of 80 x 24 cells on a 640 x 480-pixel resolution web page.

Function keys are used if the default function key labels table does not provide a label for a particular function key (defined by DDS) and the Migrator indicated that DDS has no precedence over the defaults, then DDS label is kept. Only when both DDS and the default table have labels, AND they are different, that *DDS Overrides* resolves the conflict, i.e. DDS takes precedence over the label defined by the GamePlan table.

.Net Print File Preference Directives

The .Net Print Files Preference Directives set directive defaults for printer files and printer o-specs.



Margins Group

Units: The measurement in which to view or set the margin information; either in English [Inches] or Metric [cm]. The default is English (Inches).

Margins: The two margins Monarch is concerned about are the left and top margins. The legacy report layout dictates how much data will print horizontally and vertically (the latter controlled by page overflow). The printer's physical limitations will ultimately take effect and will vary from printer to printer. A margin value of ***None** is used to indicate that the minimum allowable margin settings for the printer are to be used.

Overflow: When fetching for overflow, enter the distance starting from the bottom of the page. To indicate the fact that the measurement starts from the bottom of the page, the value is made negative. For example, -1.5 inches would mean that the overflow area starts when the printer head is 1.5 inches away from the end of the page. See "Handling Page overflow" in Chapter 10 for more information.

Orientation Group

The orientation (*portrait* or *landscape*) of the characters to be printed. When using *Landscape*, the print head will print as if the paper were rotated 90 degrees counter clockwise.

General Group

The conversion factors mentioned above (*Average char width* and the *Line Height)*, are the result of selecting a .NET Font with a specific size. As fonts or sizes are entered, Monarch computes the average width and height of the font as close as possible to the width of a character if printed using the default printer installed on the Migrators' computer.

You can override the resulting *Average char width* and *Line Height* by entering new field values. As long as the cursor is not re-positioned on the *Name* or *Size*, Monarch will not change the values entered.

Paper Kind Group

Select any of the listed .NET Paper kinds. Monarch will show the physical width and height of the corresponding paper as reference. If none of the Paper kind dimensions are suitable, it is possible to enter "*Custom*". If *Custom* is selected, Monarch changes the *Width* and *Height* boxes to edit boxes so that a custom width and height may be entered.

Selecting a Paper Kind has no effect on the positions computed by Monarch for the fields and constants that make up the report.

Insert "/Error Please set printer name" in Open Statements

Print files are explicitly opened to allow setup of print overrides. Use this default option to have a line inserted in the migrated code reminding the Migrator to set up a few properties before opening the file. The /Error compiler directive will stop the compilation and prompt the user to check the code.

This is a code snippet generated by Monarch when the selection is checked:

```
BegConstructor     Access(*Public)
     .
   Open AFFMSTPP DB(MonarchJob.PrinterDB)
/Error Please set property "Printer" (or verify it, if already set) for this .Net
Printfile.
EndConstructor
```

This line of added code will prompt the Migrator to replace the /Error line with code such as:

```
BegConstructor    Access(*Public)
   .
   Open AFFMSTPP DB(MonarchJob.PrinterDB)
   AFFMSTPP.Printer = "HP LaserJet"
EndConstructor
```

The following defaults are mutually exclusive:

- **Use this printer name in migrated .Net Print files:**

When this option is selected, the name of the printer indicated is burned-in into the new DataGate Print File. Use this technique when you want specific reports sent to their respective department printers.

- **Generate code to set the "Printer" property in migrated program, using:**

When this option is selected, the name of the printer indicated is used as the value of the Printer property of each print file right before it is opened. Setting the property overrides any burned-in printer name the file may have.

- **Don't send output to printer, instead leave manuscript in pathname:**

It may be desirable to queue all reports into a specified directory in the server so that other program may process them later. This represents the full path (directory plus name), and may contain the following replacement values:

Wildcard	Replacement value	Comments
{0}	Declared name	For example QPRINT
{1}	Job Name	Current value of MonarchJob.PsdsJobName
{2}	Job User	Current value of MonarchJob.PsdsJobUser
{3}	Job Number	Current value of MonarchJob.PsdsJobNumber

Note: *The .apm extension is associated with ASNA DataGate Renderer application by default. When you double-click one of these files, you can see a Print Preview on the server and from there send to a printer.*

Database File Preference Directives

Set these directives to suppress the source discovery for database files or the log messages when locating sources and a database file is processed. The DDS source for database files is not used and setting these options will reduce the discovery time and the number of log panel messages.

RPG Agent Migration Directives

The **RPG Migration Directives** contain variable options and direct how the migration of legacy source to AVR is to be handled by the RPG Agent.

Indent Character

The two mutually exclusive options specify the code indention character. A single *Tab Character* or the number of *Spaces* indicated is used per indent character.

Op Code and Keyword Case

The three mutually exclusive options indicate the desired 'case' for the migration of op codes and keywords. *Mixed* is a combination of upper and lower case characters, *UPPER* indicates all uppercase, and *lower* indicates all lowercase.

Preserve sequence data as line comment

Check this box to retain the legacy source sequence number in the migrated code as a line comment.

Replace compare operators with symbols

Check this box to have compare operators replaced with the equivalent symbol in AVR i.e.: EQ, NE, LT, GT, LE, GE becomes +, <>, <, >, <=, >= respectively.

Convert to Assignment

When checked, any math operation code (ADD, SUB, DIV, MULT, ZADD, and ZSUB) will be converted in AVR to an assignment operation. A Simple assignment operation consists of taking the value on the right side of the operator and assigning it to the variable on the left, as in this example x = 42. Math symbols (+, +, -, *, /) will be used for assigning values.

DclDiskFile ChkFmtId Value

Use *DclDiskFile ChkFmtId* to indicate if the format ID of the file at compile-time and the format ID of the file at run-time are to be compared. Choices are:

1. ***SAME*** to indicate that the compile-time and run-time format ID's will be compared.
2. ***IGNORE*** to indicate that the compile-time and run-time format ID's will not be compared.
3. ***MAP*** to indicate that the compile-time and run-time format ID's will be compared but also accommodate additional or changed non-key fields regardless of where they are defined in the new format.

Subroutine code generation

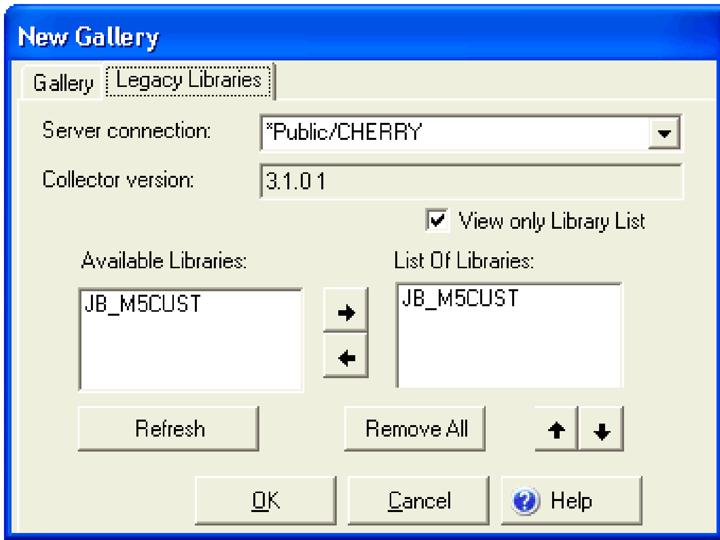
Use *Add banner on subroutines* to include a "banner" before the body of a subroutine.

Calling convention indicates your preference on how your Subroutine calls are to be converted.

Creating a Gallery

To Create a Gallery, select *File - New - Gallery* from the toolbar menu. A dialog will prompt you for the name of the Gallery and the database where to store it. You can select to store the Gallery in a local database on your PC or on a Server (which is running DataGate). The name of the Gallery should be representative of what it will contain. If you plan to store the Gallery on a System i database, then select a name no longer than 10 characters. ***Remember, a Gallery becomes a Library in the selected database.***

The following shows the dialog used to create RPGALLERY on database *****Public/CHERRY.



Hint: *Use DataGate Database Manager to create new database names.*

In addition to stating the name and location of the Gallery, you provide the list of libraries whose objects will populate the Gallery. As mentioned earlier, a Gallery is a collection of detailed information about OS/400 objects residing in a set of Libraries on a System i.

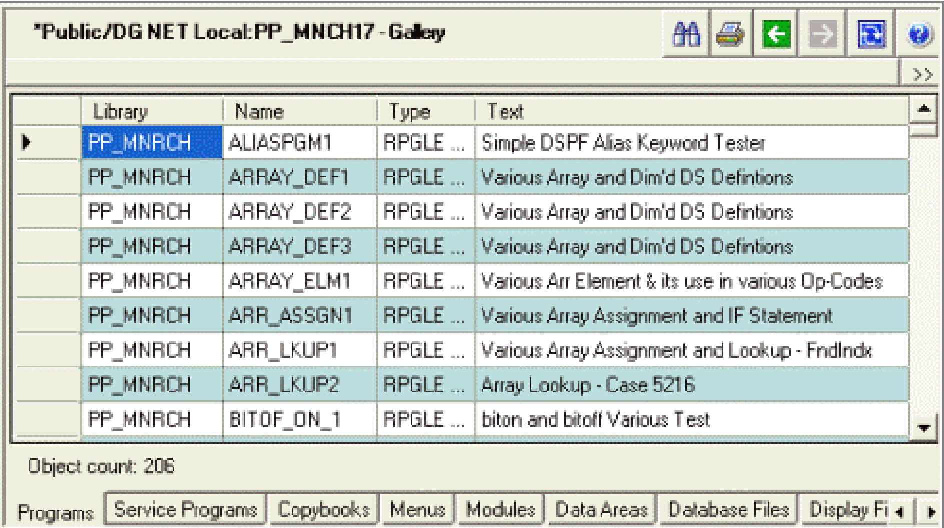
The second tab in the New Gallery dialog is *Legacy Libraries*. In this tab, you will select a connection to a System i via a DataGate database name and a list of libraries from that system. It is necessary to have the ASNA Monarch Collector Program installed on the selected server.

Once satisfied with the data entered and the *OK* button is clicked, ASNA Cocoon creates the Gallery by invoking the Collector to obtain the information for each object found in the selected libraries. This process can be lengthy depending upon the number of libraries and objects in those libraries. A working dialog box will show the progress while the collection process is running.

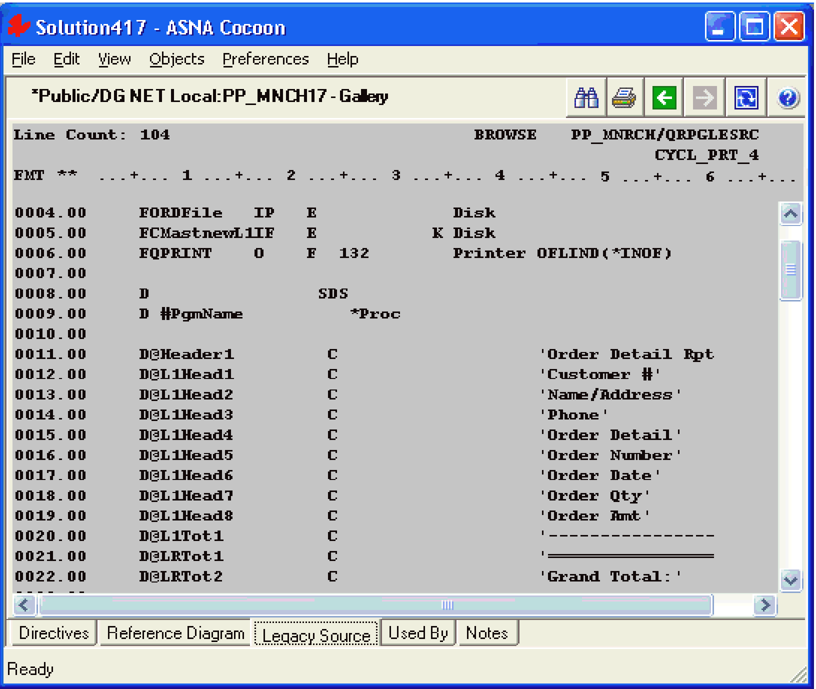
The following figure shows the library JB_M5CUST being selected from the *Public/CHERRY server connection.

Gallery Objects

After the collection process finishes running, you can view all the objects in the Gallery using the tabs in the properties panel (shown below).

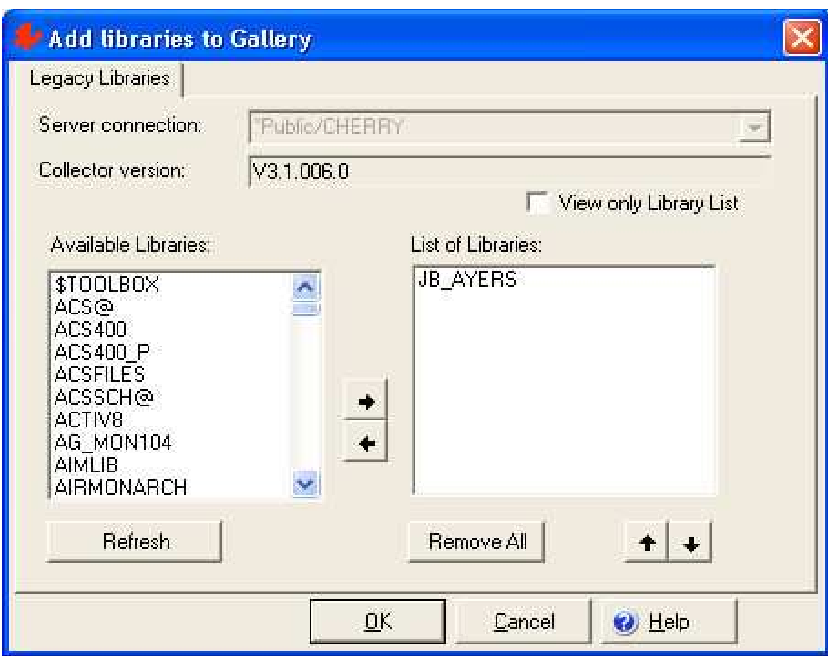


When you select an object from the Solution Explorer window, such as a program, a set of tabs in the properties panel on the right allows you to view additional information on the selected object. The set of tabs shown is dependent upon the object type. However, there are three tabs always present: **Directives**, **Used By**, and **Notes**. Many object types also have a **Legacy Source** tab as shown in the example in the next figure.



Adding Libraries

Under some conditions, it may be necessary to add libraries from the same server after the Gallery is initially created. The *Add libraries to Gallery* dialog box shown below allows you to chose the libraries to add (from the available libraries).



Dealing with the Source

It is common for the source to be moved to different libraries or even different files during the lifespan of an application.

Finding the source for a particular object can be tricky. When an object is created from source, like programs and files, OS/400 keeps track of the location and name of the source file in the object itself. When an object is added to a Gallery, the source location and name are registered with the object, along with the name of the OS/400 system where the object was found. In addition to locating object sources, copybook sources must also be located.

The mechanism Cocoon employs for locating the object source is based upon a match with the Original location that points to the target Database, Library, File, and Member where the source now resides.

The **Source Mappings** tab is used to establish the database connections used to discover sources, a library list for the discovery of copybooks, and a table to maintain an 'original to new' location relationship for locating object source.

Database Connections

This section of the Source Mappings tab contains a directive to establish the database name to use to discover sources in the Gallery and a directive to indicate the library list to locate copybook sources in that database.

Any *Database Name* entries established here are used in the drop-down box for selecting a *Database Connection* in the Source Mappings table.

The **/COPY** on OS/400 allows a fully qualified member to be specified by providing the Library, File, and Member names. It is also possible to omit the Library and/or File names. When the File name is omitted, QRPGSRC is used for RPG/400 and QRPGILESRC for RPG ILE. The discovery process follows the same convention.

When the Library is omitted, then the Library List is used by the RPG compiler on OS/400 to locate the (File/) Member. To find the proper equivalent in the discovery process, it is necessary to provide a library list identical to the one that would be used by the compiler on the System i. This defaults to the libraries used to create the Gallery but it might not be in the proper order or there might be additional libraries that must be included containing sources used by /COPY.

For the *Copybook Library-List,* you can enter a list of libraries (space separated) or enter *DBNameLibl to use the list of libraries established for the database name.

Source Mappings Table

This section of the Source Mappings tab is used to establish an association between the original source locations and the database name to get to the source. This table is important because it allows you to inform Cocoon of the **new** names where the sources can be found for particular objects or groups of objects.

The **Original (Registered) Source Location** information is:

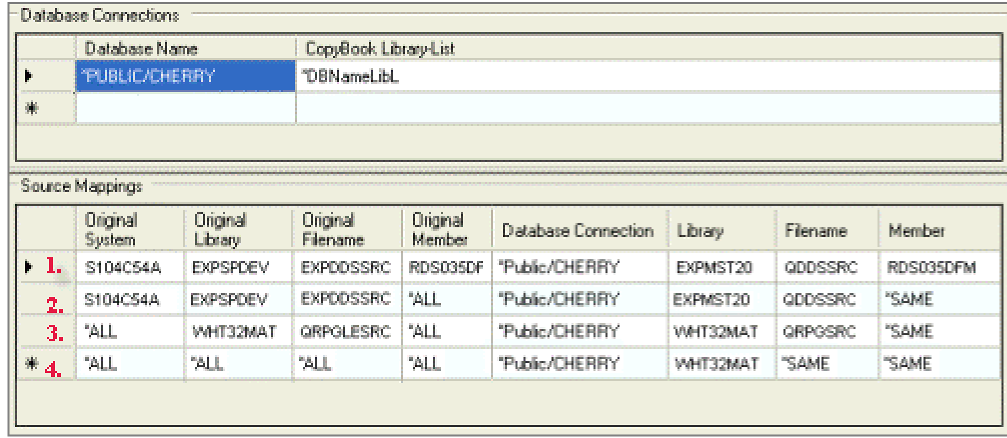
- System Name
- Library Name
- File Name
- Member Name

Since DataGate is used by Cocoon to get the source file member, it is necessary to map the original System Name to a DataGate database connection pointing to the OS/400 System where the sources can be found. This database connection should use a User Profile with enough authority to read the source's member records.

Entries in the Source Mapping table can be very generic, as when only a system name is mapped to a database connection. The value ***ALL** serves as a placeholder when entering generic entries in the table. Using ***SAME** for the target source location can be considered to mean the *SAME* Library, File, or Member.

Entries in the Source Mapping table can also have very specific mapping, such as an individual member of a file in a Library of a system to a specific Library/File/Member in a database name.

The following figure shows the Source Mapping tab.



Database Connections

Database Name	CopyBook Library-List
*PUBLIC/CHERRY	*DBNameLibL

Source Mappings

	Original System	Original Library	Original Filename	Original Member	Database Connection	Library	Filename	Member
1.	S104C54A	EXPSPDEV	EXPDDSSRC	RDS035DF	*Public/CHERRY	EXPMST20	QDDSSRC	RDS035DFM
2.	S104C54A	EXPSPDEV	EXPDDSSRC	*ALL	*Public/CHERRY	EXPMST20	QDDSSRC	*SAME
3.	*ALL	WHT32MAT	QRPGLESRC	*ALL	*Public/CHERRY	WHT32MAT	QRPGSRC	*SAME
4.	*ALL	*ALL	*ALL	*ALL	*Public/CHERRY	WHT32MAT	*SAME	*SAME

Source Search Algorithm

Referring to the figure above, the following narrative will help illustrate how mapping works:

1. The Original Member *RDS035DF* of the Original Filename *EXPDDSSRC* in the Original Library *EXPSPDEV* that resided on the Original System *S104C54A* is now found as the source Member *RDS035DFM* of Filename *QDDSSRC* in the Library *EXPMST20* that resides on Database Connection **Public/CHERRY*.
2. *All* Original Members of the Original Filename *EXPDDSSRC* in the Original Library *EXPSPDEV* that resided on the Original System *S104C54A* is now found as the *Same* Member of Filename *QDDSSRC* in the Library *EXPMST20* that resides on Database Connection **Public/CHERRY.*
3. *All* Original Members of the Original Filename *QRPGLESRC* in the Original Library *WH32MAT* that resided on *Any* system is now found as the *Same* Member of Filename *QRPGSRC* in the Library *WH32MAT* that resides on Database Connection **Public/CHERRY*.
4. *All* Original Members of *Any* Original Filename in A*ny* Original Library that resided on *Any* system is now found as the *Same* Member of the *Same* Filename in the Library *WH32MAT* that resides on Database Connection **Public/CHERRY*.

Note: *You can also include the wildcard values * and ? where * represents one or more characters and ? represents one character. If you assign multiple target locations to the same Original Member, a message will appear in the log panel and the duplicate entry will be removed.*

Source Search Sequence

The position of the mapping entry in the table is irrelevant. The Original Location matching is performed in the sequence as described in the table below where "**Any**" is ***ALL** in the Source Mapping table and "**Match**" represents the **Same** original name.

Original System	Original Library	Original File Name	Original Member
Match	Match	Match	Match
Match	Match	Match	Any
Match	Match	Any	Match
Match	Match	Any	Any
Match	Any	Match	Match
Match	Any	Match	Any
Match	Any	Any	Match
Match	Any	Any	Any
Any	Match	Match	Match
Any	Match	Match	Any
Any	Match	Any	Match
Any	Match	Any	Any
Any	Any	Match	Match
Any	Any	Match	Any
Any	Any	Any	Match
Any	Any	Any	Any

*LIBL

The ***LIBL** tab reflects information for locating legacy source object references. The tab contains three sections:

1. One to locate Message Files and Externally Described Files
2. One to locate objects *LIBL references
3. One to locate Data for Files and Data Areas

To locate Externally Described Files and Message Files (EXT. FILES)

Several constructs on OS/400 development involves the use of a reference file as a kind of dictionary whose definitions are used while defining other files, formats, fields and data structures. These constructs are found in DDS file definitions and in RPG data structure definitions.

Typically, when the reference file is used, only its name is provided, omitting the library where it lives; the corresponding compiler (DDS or RPG) use the library list current at compilation time to find the file. When these constructs are being migrated, it is also necessary for Cocoon to find these reference files. You must provide a DataGate database name pointing to the server where Cocoon can find the files used as reference files and any message files. This database name must also have its library list set in a similar manner as would be established at compile time on the iSeries.

You can also specify a different library list to be searched when locating externally described DDS files when the REF keyword is encountered.

To locate Objects Referenced by Programs, Service Programs, and Copybooks (OBJECTS)

This library list is used to resolve *LIBL references to locate objects for Exhibit Graphs, to create GamePlans, for Call Graphs, and for Object usage graphs. You will need to specify the order in which the libraries are used, which is significant because when an object is referred to by library list (*LIBL) and there are multiple objects with the same name, Cocoon will select the first one found according to the sequence of libraries given. This is especially significant for program calls. When Cocoon assembles the set of objects to be included, the program-call tree is traversed and the programs in the tree are those found according to this library list.

To locate Data for Files and Data Areas (DATA)

You must provide a DataGate database name pointing to the server where Cocoon can find the data files and data areas. *DBNameLibl can be specified when the library-list defined in DataGate is the library list to be used.

Checking the Source Mappings

After setting the Source Mappings table and ***LIBL directives, one way of finding out if they are set up correctly is to view the Gallery **Missing Sources** as shown below. You can also view the **Out of Date Objects** for the Gallery showing those objects where the object date is older than the source date, i.e. the source has been changed and not recompiled.

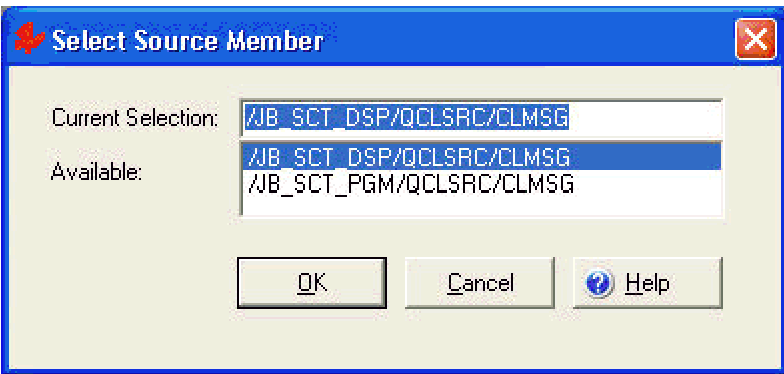
Locating Object Sources

To locate sources, select *Object - Locate Object Source* from the toolbar menu while having the Gallery selected. The name of each object for which the source could not be found is displayed in **red** in the solution explorer window.

> ***Hint:*** *To suppress the database file messages during the* ***Locate Object Source*** *process, select the checkbox found in the* ***Database Files tab*** *in the* ***Preferences - Default Directives*** *found on the toolbar menu.*

For missing source, click its Legacy Source tab and you will find the Original Source Location and the location used in the unsuccessful search for the object.

Notice the **Find Member** button that can be used to help locate object sources for you. Members found that might be the missing source will be displayed allowing you to select the member to be added to the entries in the Source Mapping table.



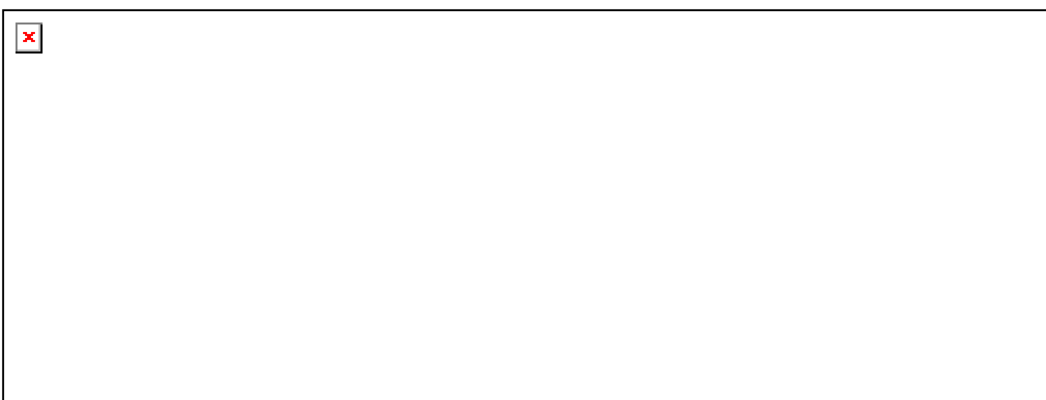
Once you select *OK*, the entry is added to the Source Mappings table as shown below.

Changing Source Location

In lieu of the *Find Member* button, you can also correct the Source Mappings table and *LIBL directives by editing an existing entry or by adding a new entry for any missing

objects. In addition, you may find it necessary to change the source location for an object. First, select the folder in the solution explorer window containing the offending object. Then, select the offending object in the properties panel and select *Change Source Location* from the context-sensitive menu.

The dialog box presented then allows for the entry of a new location for the source.



Missing Objects

The *Missing Objects* tab can also provide information on references to objects that are missing.

Synchronizing Single-Module Programs

For Galleries with single module programs, the *Synchronize Single-Modules Programs* menu option takes a single module program and binds it to a module when the legacy source is the same for both objects.

This effectively treats the module and the single-module program to which it is bound as a single object. In addition, any artifacts discovered will also be treated as a single object even though they will appear under both the single-module program and the module.

Because the program source and module source may be located at different location, this process cannot run when the Gallery is initially created. You must manually execute this synchronization AFTER the source mapping table is correct AND the sources have been located successfully.

Discovering Artifacts

Artifacts are separate chunks of code used in one or more programs collected during the discovery process and defined by their content as:

- Copybooks
- Printer O-Specs
- Data Structures
- Procedures

Copybooks

Most RPG compiler directives (like /TITLE, /EJECT, /SPACE) are ignored by the RPG Agent. The most notable exception is /COPY.

As far as the RPG Agent is concerned, a Copybook is **always** read into memory. All of its specifications are inserted at the point where the /COPY is defined and parsed as if it was part of the program. During the discover process, all the RPG Program sources are searched for /COPY.

Even though Copybooks are not strictly OS/400 objects, Cocoon treats them as if they were objects to facilitate the task of setting Directives on how to treat individual Copybooks. Copybooks require special considerations since they can also contain other artifacts i.e. Data Structures, Procedures, and embedded Copybooks.

You may see a single copybook:

1. In the **Copybook folder**, that contains all copybooks for the Library or GamePlan, including any embedded copybooks.
2. In any **Copybooks artifact sub-folder** under all object that references that copybook, including the parent copybook.

If you have not already done so, Cocoon may prompt you for a valid *default location* where the Copybook sources are to be located when migrating a GamePlan. This default location is the initial Directive for each Copybook found in the discovery process.

Printer O-Specs

ASNA Visual RPG supports printing using externally defined files but does not directly support printing using O-Specs. It is then the job of Cocoon to migrate printer O-Specs into externally defined Printer Files. Since creating Printer files requires the ability to name them and select the database to locate them, it is convenient to localize the O-Specs used in printing that are embedded in RPG Programs.

When Printer O-specs are defined within a copybook, the O-spec will be shown as an artifact for EACH program, service program, and module in which it is used and **NOT** as an artifact for the copybook in which it is defined.

Before discovering Printer O-Specs, the .Net Print File Directives defaults should be established as the initial directives for the Printer O-Specs discovered.

> ***Note:*** *There must be a valid printer driver installed in order for Cocoon to validate the directives as acceptable for the defined printer and to calculate point size, average character width, and line height for the specified valid font. This also includes the Margins, Orientation, and Paper Kind.*

Once the discovery process is complete, all Printer O-Specs appear in the Printer O-Specs artifacts sub-folders under the objects that refer to them. The Directives for each Printer O-Spec can then be set independent of the default settings.

See Migrating Print Files in Chapter 10 for more detail.

Data Structures

ILE RPG allows you to define a data structure as an area in storage and to define subfields in that area.

You can use a data structure to:

- Re-define the same area using different subfield definitions
- Define a data structure the same way as a record
- Define multiple occurrences of a set of data
- Group non-contiguous data into contiguous internal storage locations
- Reference all the subfields as a group by the name of the data structure
- Reference an individual subfield by its name

In addition, there are four special data structures, each with a specific purpose:

1. A data area data structure
2. An indicator data structure
3. A file information data structure (partially supported)
4. A program-status data structure (partially supported)

Data structures can be either program described or externally described. The RPG Agent converts all externally described data structure to program described by using the external

name to locate and extract the description of the data structure subfields. Monarch identifies and collects these data structures as artifacts during the discovery process. Any error detected in a data structure is listed in the **Artifact Errors** tab.

Data Area Data Structure

Data area data structures, as in all data structures, have the type character. The data area and data area data structure will have the same name unless your data area has been renamed using the *DTAARA DEFINE operation code or the DTAARA keyword.

Indicator Data Structure

An indicator data structure is identified by the keyword INDDS on the file description specifications. This data structure is used to store conditioning and response indicators passed to and from data management for a file. This lets you associate a data structure name with the indicators for a workstation or printer file.

ASNA Monarch supports Indicator Data Structures for externally described printer files and externally and program-described workstation files.

The data structure can be defined as a multiple occurrence data structure and the subfields can contain arrays of indicators as long as the total length does not exceed 99.

File Information Data Structure

A file information data structure (defined by the keyword INFDS on a file description specifications) can be specified for each file in the program and contains predefined subfields that provide information on a file exception error that occurs.

ASNA Monarch only supports certain fields and keywords for the File Information Data Structure for printer and workstation files. Monarch's implementation of the File information Data Structure is not a true data structure per se, but a list of read-only properties that contain the values of their respective status fields. Refer to Status Data Structures in Chapter 8 for more information.

Program Status Data Structure

A program status data structure provides program exception information to the program. The PSDS is defined in the main source section; therefore, there is only one PSDS per module.

The location of the subfields in the PSDS is defined by special keywords or by predefined From and To positions. ASNA Monarch only supports certain fields and keywords.

Monarch's implementation of the PSDS expands beyond the simple support of migrating the data structure. A "Populate Program Status Data Structure" section in the constructor sets the fields and properties from the fields in the data structure. Refer to Status Data Structures in Chapter 8 for more information.

Procedures

A procedure is a self-contained program that can be called to perform repeated or shared tasks and then return to the caller. Procedures can include both host language statements and SQL statements.

In most cases, a procedure needs some information about the circumstances in which it has been called. This information consists of variables, constants, and expressions that are passed to the procedure. A procedure may return a value to the calling program.

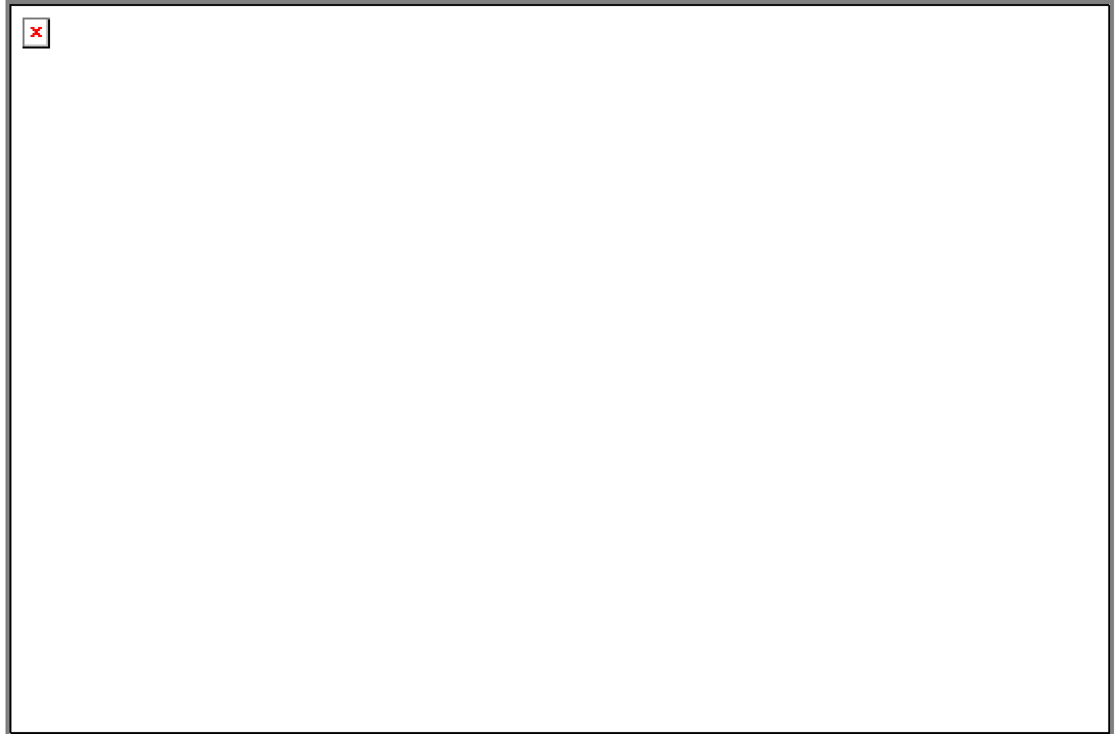
A procedure:

- Names the procedure.
- Defines parameters and their attributes. A parameter represents a value that the procedure expects. The procedure's declaration defines these parameters.
- Gives other information about the procedure including the return value.

'Values' are supplied to a procedures parameter when you call the procedure. It is left to the calling program to perform handling of any return value.

You can set the procedure directives and view additional information on a procedure using its directives tab. This view (shown below), will show the procedure name, information about the return value, the list of parameters, and any errors detected.

Notice there is an external reference that will be checked if the keyword EXTPGM(name) or EXTPROC(name) is given for the procedure.



Refer to the procedure directives in the Cocoon User's Guide for more detail.

This page is intentionally left blank.

Chapter 3 - Working with Exhibits, Sections, and Clusters

Partitioning an Application

An OS/400 application is composed of several Programs that call each other to form a call graph that can be represented with a call tree. Conceptually, any part of the tree can be grouped into ***Subsystems*** of the Solution. A Subsystem that starts at the root of the tree can be as small as only the root or as large as the whole tree. At the other end of the tree, a single leaf can be considered a Subsystem or any of the branches can also be considered a Subsystem. Subsystems can be either ***interactive*** or ***non-interactive***. The root Subsystem is considered an Application and any other Subsystem (branch or leaf) is considered a ***utility Subsystem***. Utility Subsystems might be shared across Solutions.

The first challenge a Migrator faces is to decide how to partition the usually large collection of objects into smaller more manageable groups, which will eventually become Visual Studio .Net assemblies (.DLL).

Monarch starts with a Gallery and requires the user to select objects from this collection to form GamePlans. The selection process may be difficult since it requires in-depth knowledge of the object relationships, particularly program references to other objects including other programs.

Cocoon assists the Migrator by providing tools to partition the Gallery in an **"Exhibit"** containing one or more **"Sections"** that are further divided into even smaller chunks called "**Clusters**".

Exhibits

An **Exhibit** is a container for one or more Sections and the Clusters each Section contains. Exhibit Directives are the initial defaults for the sections and clusters an exhibit can contain. You also have a **Call Graph**, **Exhibit Details** tab, **DB Files Graph**, and a **Magnitude** for your Exhibit.

You will find examples later in this chapter.

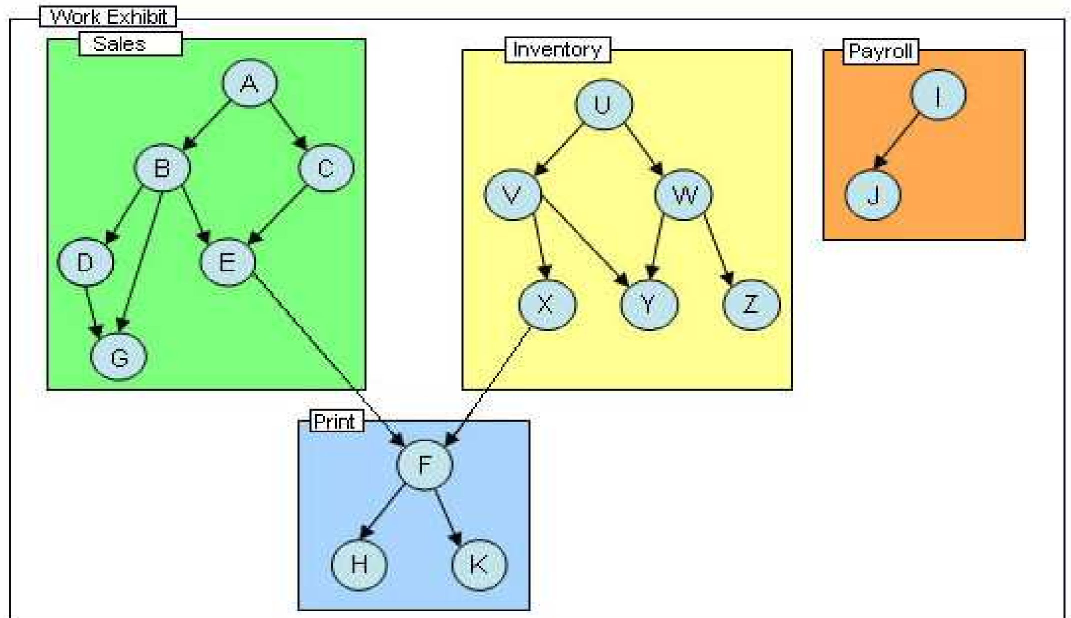
Sections and Clusters

A **Section** is a partitioning of the Gallery from the objects selected based upon the rules established by the Migrator. A Section rules definition can 'segment' the programs in your Gallery by whatever best fits your needs. A new Section is created to contain the objects that conform to the rules you have established, formed into a *natural cluster* or *single program cluster*.

A **natural cluster** is formed beginning with a single entry point program and follows the call graph to include only the programs that can be reached by following just that path to become part of that cluster.

The **single program cluster** option forms a cluster from a single program regardless of the call tree structure (performs no program clustering).

The following diagram shows **Work Exhibit** with four sections **Sales**, **Inventory**, **Payroll**, and **Print** with the section call sequence showing *Sales* and *Inventory* calling *Print*. It also shows the natural cluster call sequence where *Sales* ***Cluster E*** and *Inventory* ***Cluster X*** both call *Print* ***Cluster F***. Looking at the example, you can see other possible sections that could have been defined or clusters that could have been grouped to form an entirely different picture. Ultimately, any sections and clusters must be based on the processes performed in your particular application. More detail on how to group clusters and create sections is covered later in the chapter.



Cluster Definitions

Entry Point Cluster

Cluster with an entry-point program or service program.

Internal Cluster

Non *Entry-Point* Cluster.

Terminal Cluster

Cluster does not call any other cluster.

Single Cluster

An *Entry Point Cluster* that is also a *Terminal Cluster*.

Single Program Cluster (as defined above)

Natural Cluster (as defined above)

User-Constructed Cluster

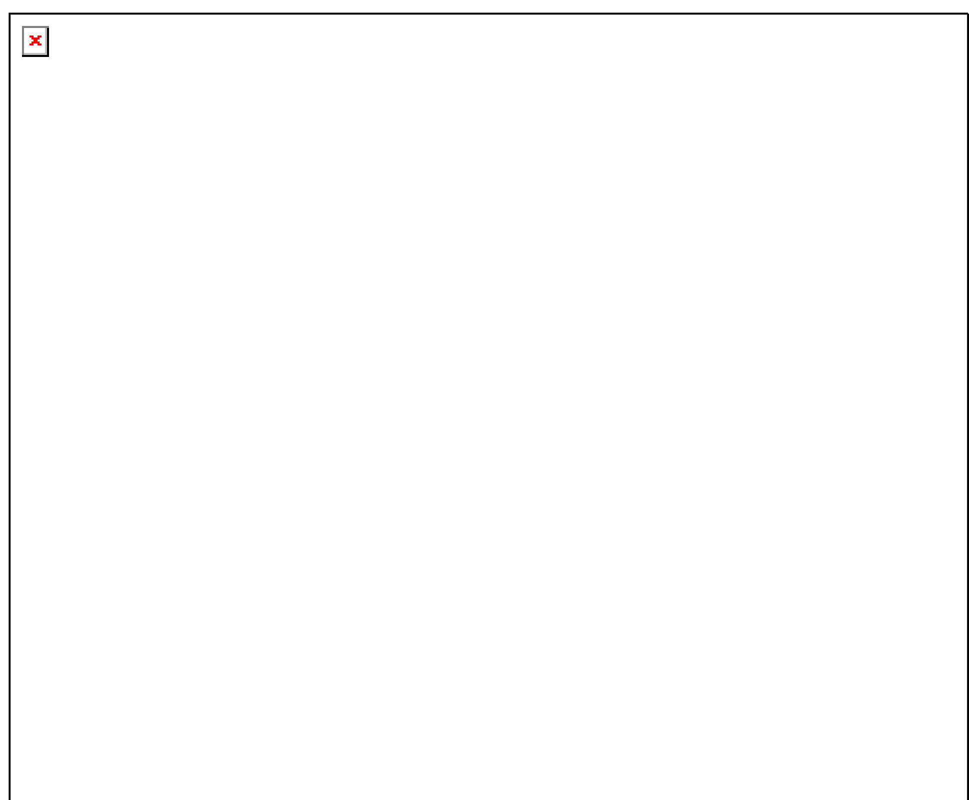
An arbitrary group of programs and service programs.

Creating Exhibit Sections

You create a Section by creating a new Exhibit then add a rules definition for each section in the Exhibit.

Create a New Exhibit

Select any Exhibit node in the Solution Tree then right-mouse click and select *New Exhibit* from the context-sensitive menu. The **Create Exhibit** dialog will display as shown in the next figure.



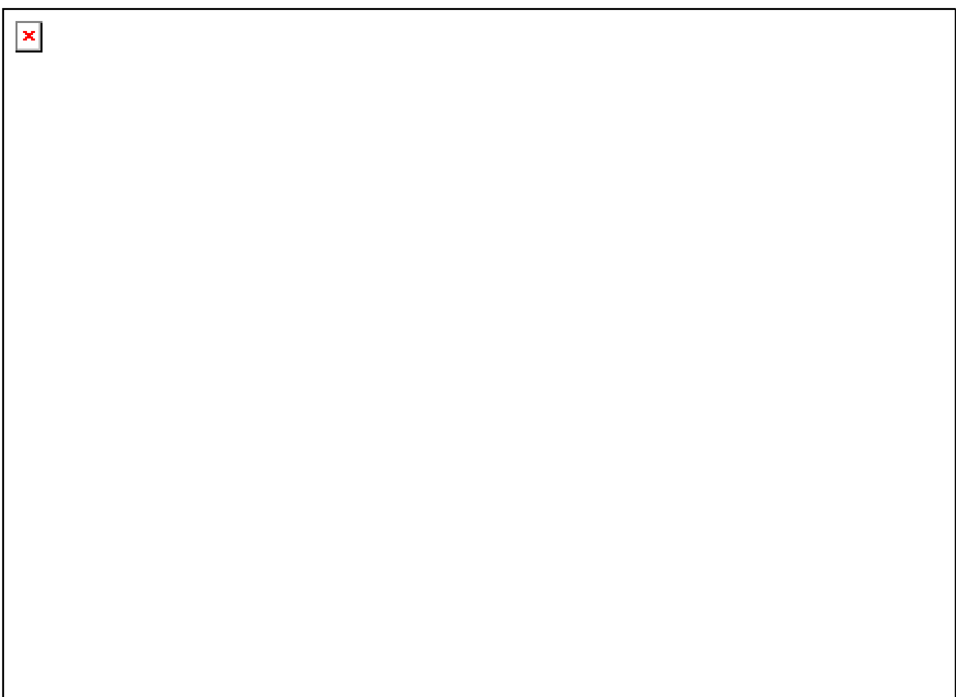
Enter *Name*, *Author*, and *Description*, and then enter a *Namespace* and a *Library List* used to discover program references. Notice the *Single Program Clusters* checkbox. Checking this option will cause each program selected by the section rules definition to create a cluster.

Two buttons, *Import* and *Export*, allow you to save section rule definitions to an Xml file and then import that file when creating another Exhibit. Use *Add* to define rules for a section, *Edit* to change rules in this section, and *Delete* to delete the selected section.

> ***Note:*** *There is always one* ****Catch All*** *section created automatically to contain all clusters in the Gallery that are* ***NOT*** *contained within the defined sections of the exhibit.*

Defining a Section

When the **New Exhibit Section** dialog appears, enter the section *Name* and *Description*. The Namespace shown will display from the Exhibit Namespace value. You can also specify a different namespace by checking the *Custom* checkbox and entering the namespace in the textbox.



The Rules grid is where you enter the rules to select or omit objects from the Section you are defining.

Section Rules

Rules entered contain an ***Action***, ***Attribute***, ***Operation***, and ***Value***. The possible values for the first three are selectable from a drop-down box (ID is an internal reference only). "***Is Regular Expression***" will be explained as we continue.

Action options include '*Select*' or '*Omit*'. *Attributes* available indicate cluster name, clusters containing a program, clusters with variable calls, etc. *Operations* can be *EQ, NE, LT, LE, GE*, etc., and will vary depending on the *Attribute*. *Value* is the value portion of the expression compared to the *Attribute* and includes ***True*** and ***False*** in some instances.

Is Regular Expression indicates if the value should be treated as a .NET regular expression. *False* indicates *Value* is a string that can contain any number of wild card characters where * is any sequence of characters, ? is any single character, and % is any digit. *True* indicates *Value* is a .Net expression. For example, ^[A-L].*$ indicates the first letter in a range from capital "A" through capital "L", or take sdp(0|1)[0-5].* that indicates the first three letters "sdp", followed by "0" or "1", and then any number "0" through "5".

There is a special *Operation* "*CalledBy"* that allows selection of any cluster "called by" the clusters previously selected by ALL preceding rules. *Action*, *Attribute*, *Value*, and *Is Regular Expression* are ignored.

> *The sequence in which each rule appears is extremely important.* ***Once a cluster is 'selected' or 'omitted' by a rule statement, any subsequent rule in the section definition has no affect on that cluster.*** *You can use the up and down arrows to place a rule in the order needed.*

Press *Ok* to complete the rules definition and return to the Create Exhibit dialog. The image below shows Exhibit "DailyWork" with six sections showing the rules definition for three of those sections.

> *The sequence in which the Sections appear in this dialog is also extremely important.* ***Once a cluster is 'selected' by a section rules definition, any subsequent section rule definitions are ignored.***

Press *Ok* to create the Exhibit.

Exhibit, Section, and Cluster Tabs

Before starting to work with sections and clusters, let's see how graph contents are colored and view samples of the tabs available for an Exhibit, Section, and Cluster. Note that some of these tabs will have context-sensitive menus available to perform some of the functions necessary to work with the panel contents. More detail will be covered later in this chapter.

Graph Coloring

The type of program, cluster, or section is reflected in its 'color' in the various graphs. The type is determined by the following:

- Any display files, then it is " ***Interactive*** ", otherwise it is " ***Process*** ".
- Any call using a variable, then add " ****VAR*** ".
- Any call to an unknown missing program, then add " ***?*** ".

Exceptions:

Clusters being grouped are temporarily shown as **"white"** until the graph is refreshed. The type Cocoon assigns to the new grouping will be re-determined as indicated above. Any unknown missing programs and service programs called will be shown as **"Unknown" (yellow)** in the **Cluster Call Graph**. A Section referenced in the **Section Call Graph** and external program references in the **GamePlan Call Graph** are always **Gray**.

Example:

Notice the coloring of the various objects in each column below. For example, Column A shows the programs as they might appear in their respective **Cluster Call Graph**, Column B shows those three clusters viewed from the **Section Call Graph**, and Column C shows that Section as it would appear in the **Exhibit Call Graph**.

Exhibit Directives

The Directives shows the migration directives for an exhibit. The *Sections* lists the section names defined within the Exhibit.

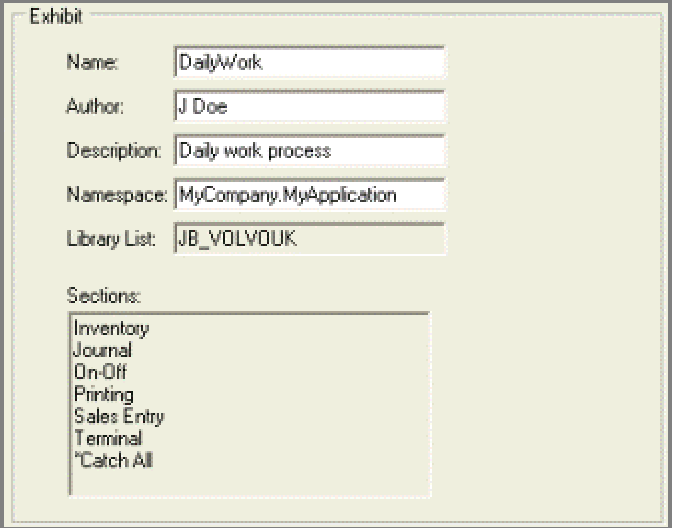


Exhibit Call Graph

The Call Graph shows the call relationship for all Sections in the Exhibit.

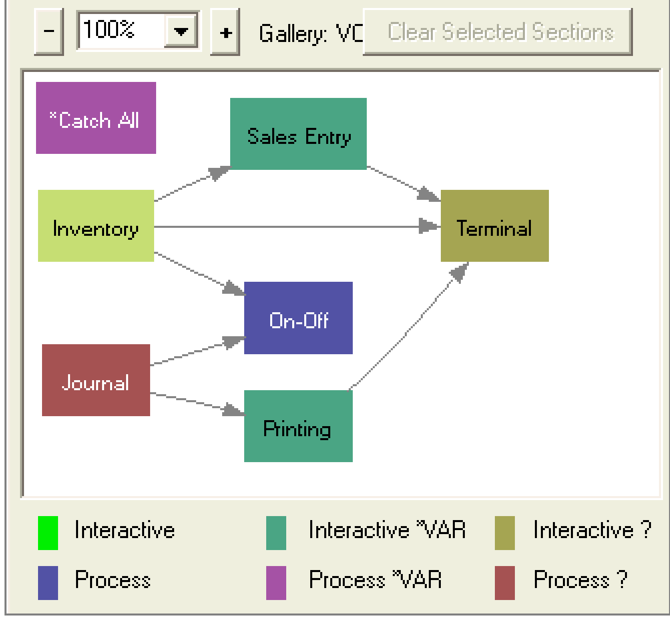


Exhibit Details (Statistics)

The Exhibit Details contains valuable information as part of the evaluation of all the clusters and programs in the exhibit. It gives the different totals to help in the overall sizing of the contents of the Gallery.

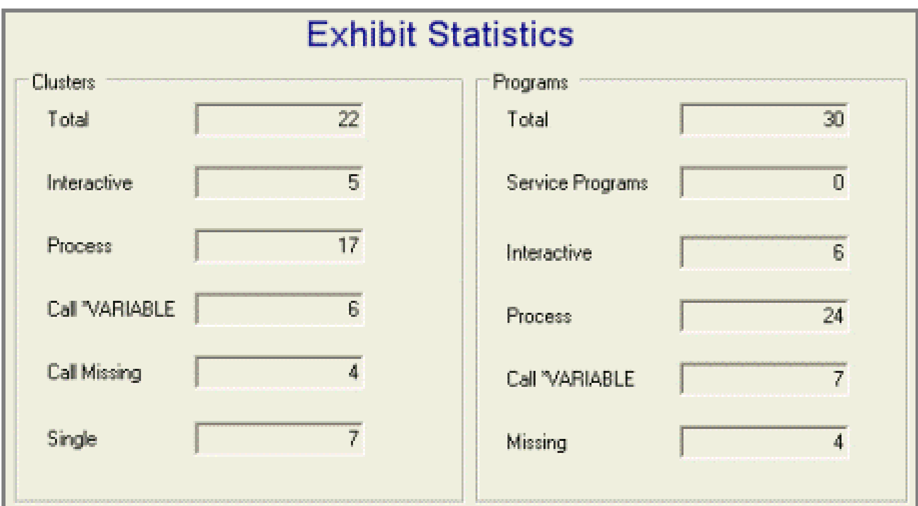


Exhibit DB Files Graph

The Exhibit DB Files Graph shows the file relationships for the database files contained within the Gallery.

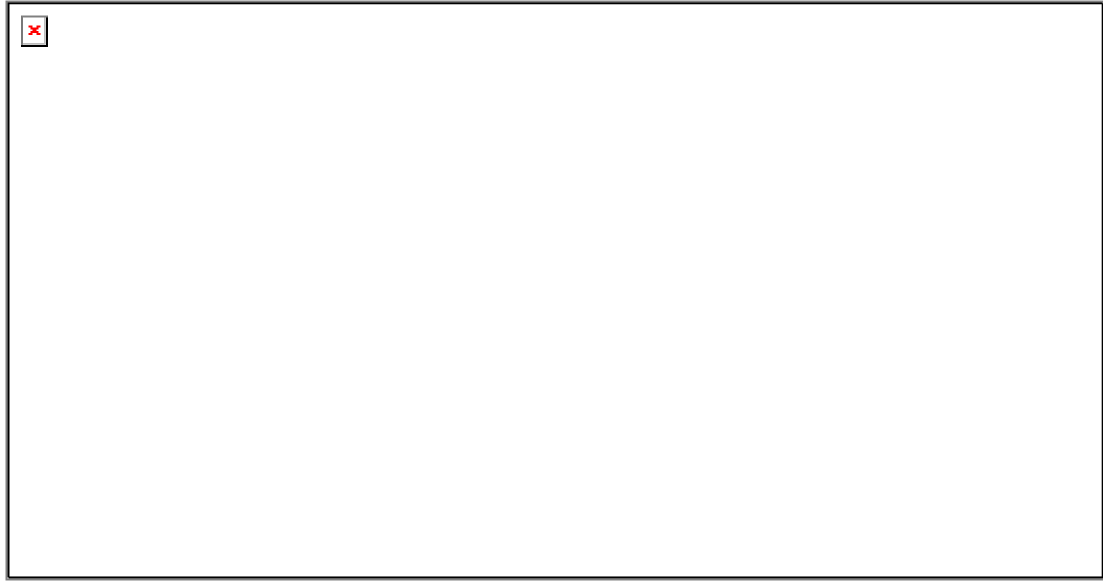


Exhibit Magnitude

The Exhibit Magnitude tab shows information on the contents of the Exhibit. A Magnitude tab is also available for a section and cluster containing just the information for the section or cluster respectively.

Section Directives

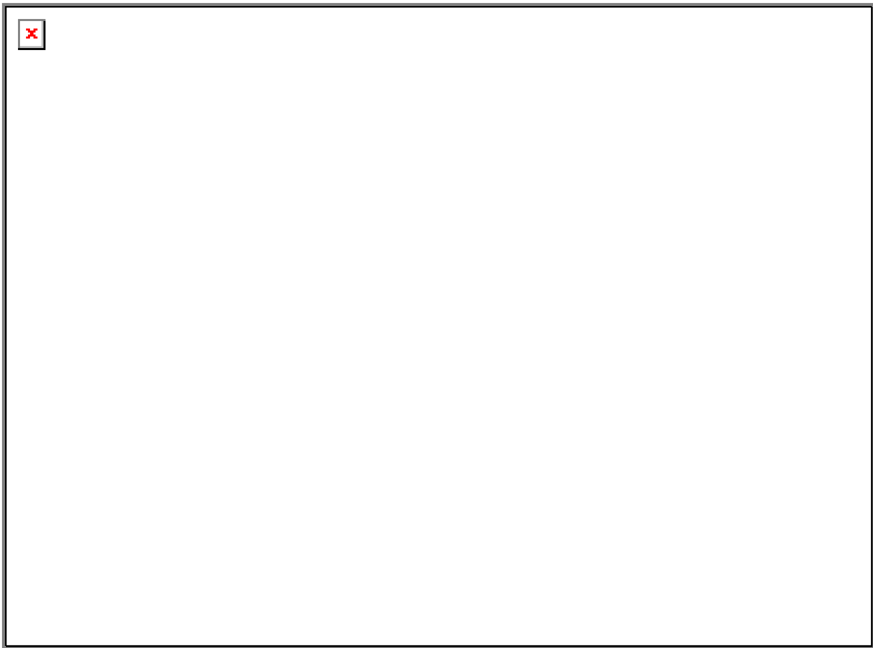
The Section Directives provide information on the section and shows the *Rules* by which the section was formed.

Section Call Graph

The section Call Graph shows the cluster call sequence within the section (inside the box) as well as the sections call sequence (outside the box).

Section Details

The Section Details provides information, similar to the Exhibit Details tab, but just for the contents of the section. Notice the checkboxes that will show when a section contains grouped cluster, external cluster, or if some cluster have been moved.

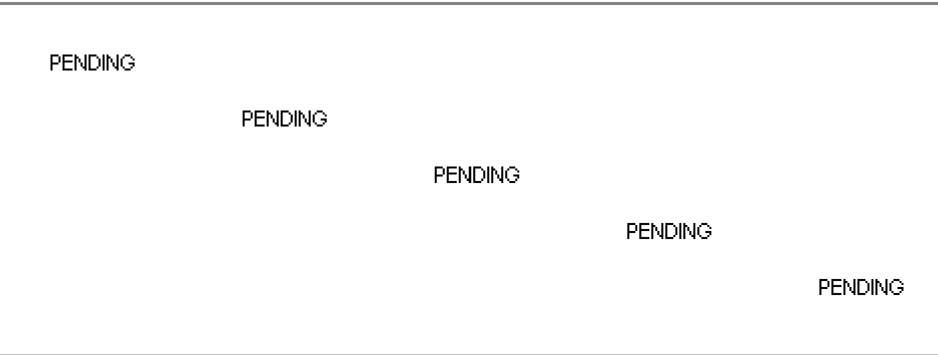


Cluster Directives

The Cluster Directives tab enables the Migrator to establish directives for a single Cluster.

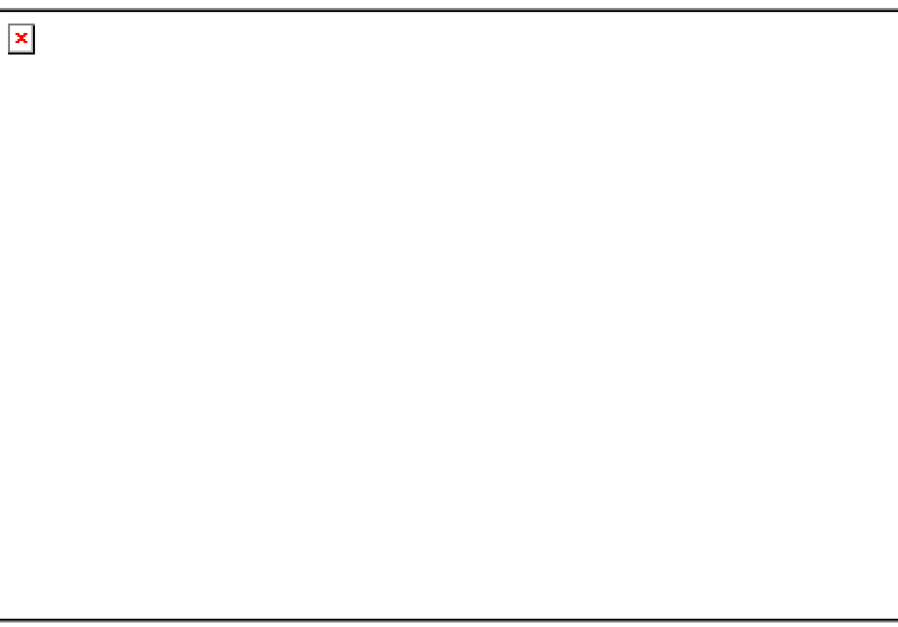
Cluster Details

The Details tab provides information about the contents of a cluster. The **Entry Points** and **Missing Objects** are in the left-hand column and the **Programs**, **Display Files**, and **Menus** are in the right-hand column.



Cluster Call Graph

The Call Graph tab shows the **program call sequence** (shown within the colored block) and the **cluster call sequence** (shown outside the colored block). In the example below, "*SL5230*" is a program while "*SL523HEC*" is a cluster.



Working with Sections and Clusters

Once you have created an exhibit with its sections, you can create new group clusters. Keep in mind that the majority of the 'grouping' occurs when defining the rules for a section as outlined earlier in the chapter. With good rule section definitions, there may be no need to do any of the re-grouping noted here.

Re-Sectioning Exhibits

Once a new exhibit is created, the Migrator can play with different scenarios and group multiple sections into a new section or even group multiple clusters in a section into a new single group cluster.

These sections may be candidates for packaging as an ASP.NET Web application, Class Library, or Console Application. Once again, knowledge of the application is required here. Take the following diagram, for example.

This system has two natural entry points, hinting at two applications: one starts at **A** and the other at **S**. Taking each in isolation could lead to some arbitrary partitioning. However,

when we take into account both Entry-Points, it becomes clear that having **Z** in some Class Library may be desirable as it has multiple entry points. For example, you could considered the following partitioning.

Copying Exhibits

There are two ways to create a copy of an existing exhibit. Using **New Exhibit** and **Copy definition for New Exhibit** from the context-sensitive menu.

New Exhibit will make an exact copy of the exhibit on which this option is selected.

> *The new Exhibit Sections **WILL** retain any re-grouping of clusters already done.*

Copy definition for New Exhibit will also make a copy of the exhibit on which this option is selected but you get the additional options of adding, editing, or deleting the rules definitions for the sections it contains.

> *The new Exhibit Sections **WILL NOT** retain any re-grouping of clusters already done.*

The **Create Exhibit** dialog box and **Edit Exhibit Section** dialog boxes that appear are the same as when adding a new exhibit and defining sections noted earlier in the chapter.

Creating Group Clusters

You use the Section **Call Graph** to create group clusters or select Clusters directly from the solution explorer window. To create group clusters from the Call Graph, you can use **Find** or simply begin selecting clusters using Ctrl+left-mouse. The selected clusters appear outlined as shown in Figure 1. You also use Ctrl+left-mouse to select clusters from the solution explorer window as shown in Figure 2.

Select **Group Cluster** from the context menu on the graph or the solution explorer window and enter the name and description for the new cluster. The new grouping will appear "white" in the Section Graph as shown in Figure 3. Press the Refresh icon to update the graph as shown in Figure 4 or select **Undo Cluster Operation** from the context-sensitive menu to cancel the grouping.

Splitting Clusters

First, select the cluster from the Section **Call Graph** or from the solution explorer window, and select **Split Cluster** from the respective menu. The **Split Cluster** dialog box appears with two columns and the cluster call graph at the bottom for reference. Enter the **Name** and **Description** for Cluster 1 and Cluster 2.

All programs are initially listed in the left-hand column (Cluster 1) when the dialog opens. Select programs from this column and press the right-arrow to move it to the right-hand column for Cluster 2. To reverse a selection, select the program from the right-hand column and press the left-arrow. Figure 5 shows a Split Cluster dialog. When completed, the split cluster will appear with a dash outline in the Section **Call Graph** until the **Refresh** icon is pressed or you select **Undo Cluster Operation** to cancel the operation.

Additional Functions

There are additional context-sensitive menu options available on the Section Call Graph and Exhibit Call Graph to assist you in working with clusters and sections.

Additional Section Call Graph context-sensitive menu options:

1) **Explore** - jump to Call Graph of Cluster.
2) **Highlight** - highlights the call sequence arrows of incoming, outgoing, or both.
3) **Properties** - view the properties of a cluster.
4) **Move to Section** - move a cluster to another section.

Additional Exhibit Call Graph context-sensitive menu options:

- **Explore** - jump to Call Graph of Cluster.
- **Highlight** - highlights the call sequence arrows of incoming, outgoing, or both.
- **Properties** - view the properties of a section.

Examples

Figure 1: Selected Clusters for New Group using the Section Call Graph

Figure 2: Selected Clusters from Solution Explorer window

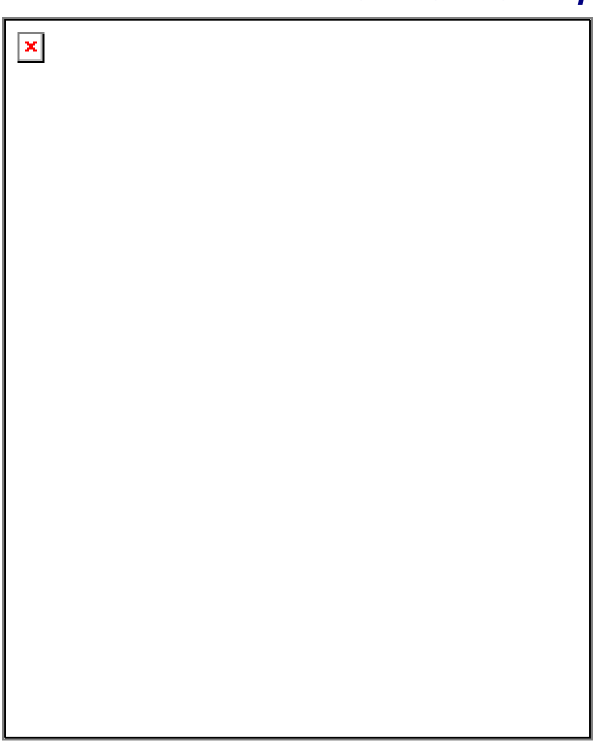


Figure 3: After Group Clusters selected, named "NewGrpTerm".

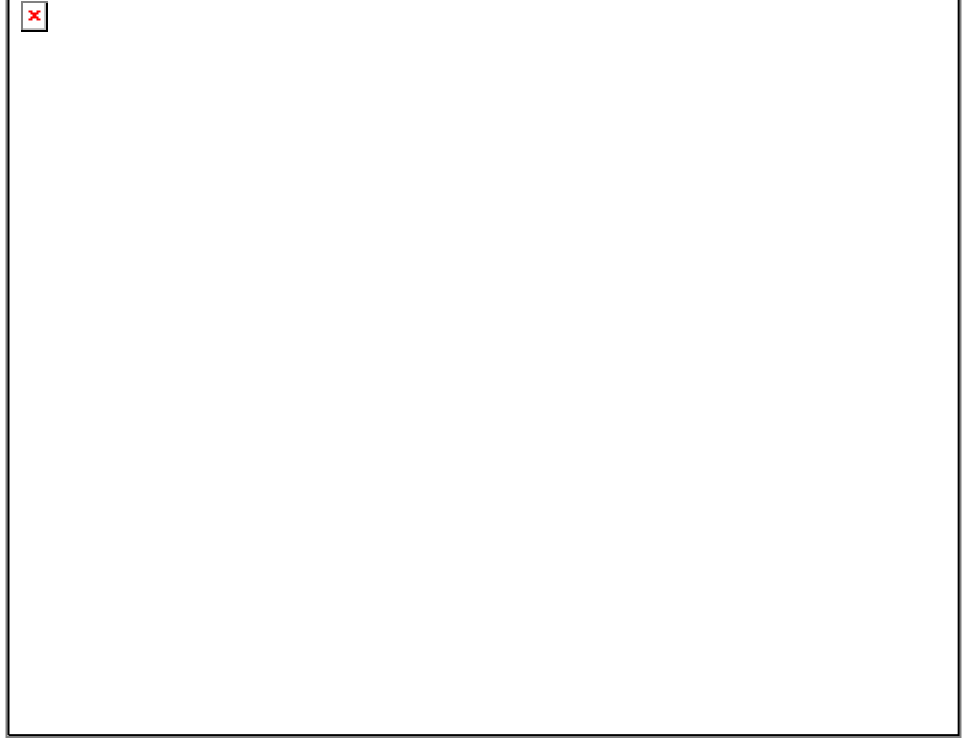


Figure 4: Section Call Graph after "Refresh".

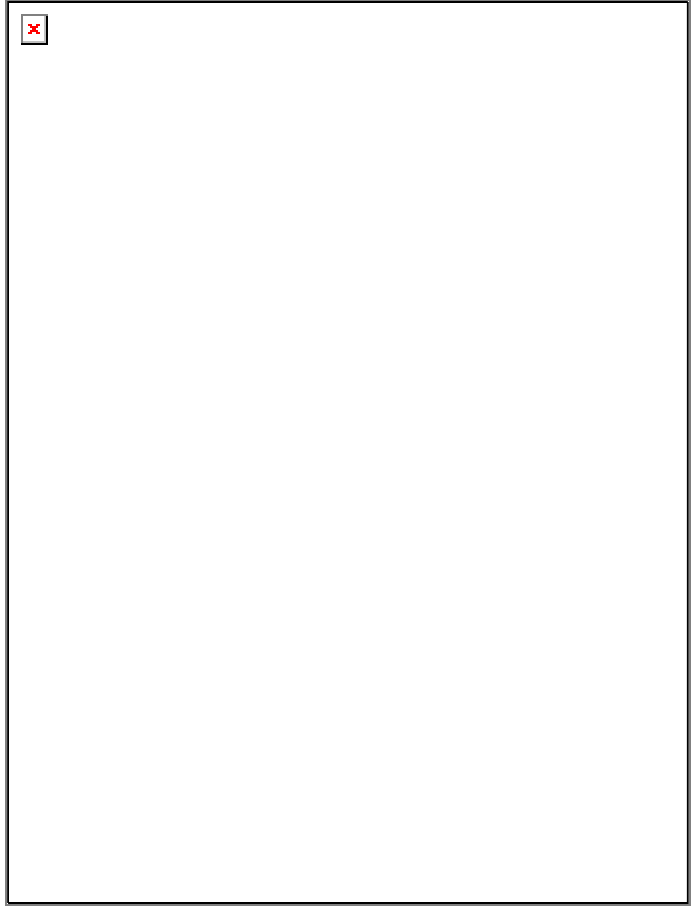
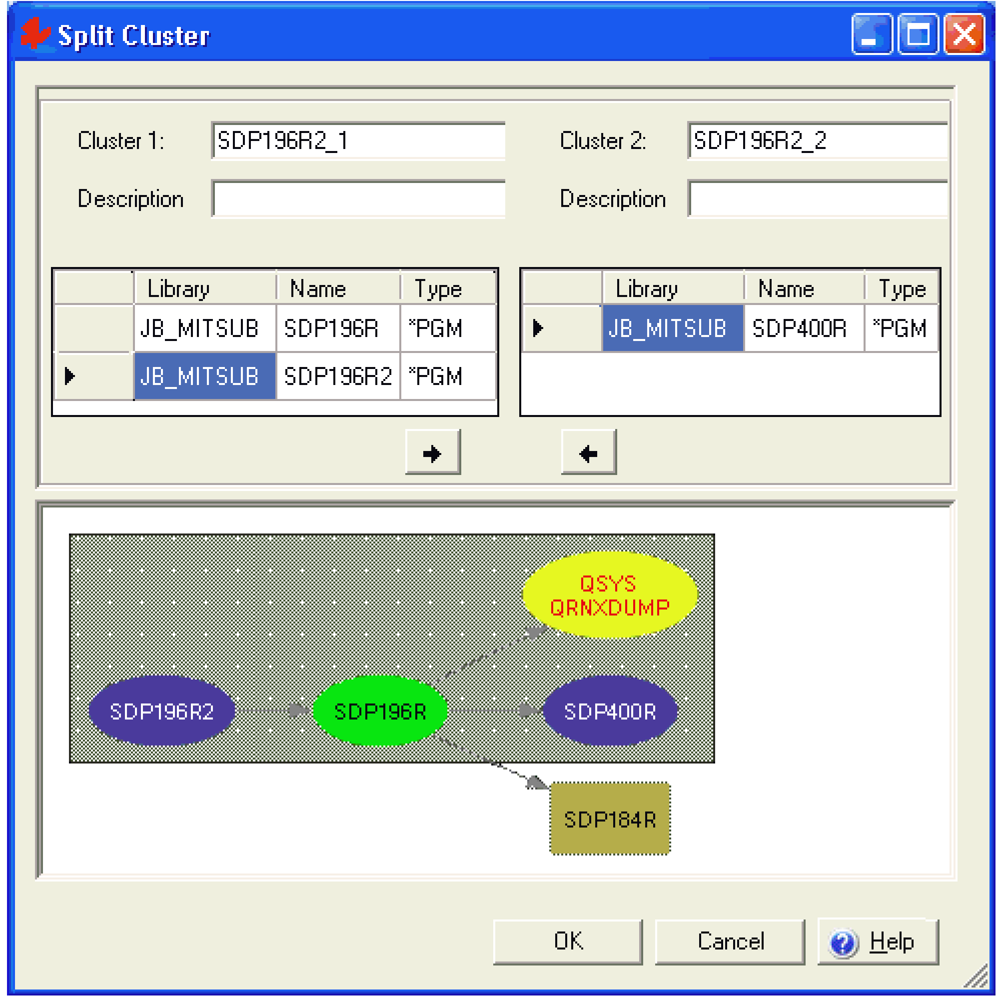


Figure 5: Split Cluster Dialog.



Chapter 4 - Working with GamePlans

Project Types

Monarch can migrate GamePlans into five different kinds of Projects (not including Data Only GamePlans), divided into **interactive** (uses Display Files), and **non-interactive** (does not use Display Files).

The Migrator chooses the Project type according to how the resulting assembly will be used. The following table defines the characteristics of the Project Types.

Project Type	Started by	Display Files	Entry Points	Caller
ASP.NET Web Application	Browser Request	Yes	1	User
ASP.NET Web-Module	Browser Request in Host Solution	Yes	No	User in Host Solution
Class Library	Call/Parm	No	1 or More	Other Programs
Console Application	Command Line or OSEXEC	No	1	User or Scheduler or Other Programs

The Migrator can also create a GamePlan with a Project type of **None** for OS/400 objects that are migrated once and do not generate any kind of code. A GamePlan that contains only Message files or Print files is a good example of this kind of GamePlan. When these Projects are migrated, they ***do not* produce a Visual Studio Project.**

Interactive

Interactive Project types have Programs that make use of Display Files. The Display Files are converted into ASP.NET pages.

ASP .Net Web Application and ASP. NET Web-Module

An ASP .NET Web Application is an interactive application that is user-initiated - that is the user will select a menu option. It is an entry-point into a subsystem. If it receives any parameters, they are typically input-only. **This Project type does not return any values**.

Feature	Value
Entry points	1
Display Files	Yes
Caller	User
Started	Browser Request

Implementation

Visual Studio Project Type:	ASP.NET Web Application
Monarch Job for compile-time:	A MyJob class is generated based on Monarch.WebJob
Monarch Job for run-time:	Uses an instance of its own MyJob
Source files comprising the Project:	vr, .aspx, .aspx.vr, Global.asax, Web.config
Files need to run the Program:	.dll, .aspx, Global.asax, Web.config, ASNA.Monarch.WebDspF.dll
Templates at:	&/Monarch 4.0/Cocoon/Templates/AspWebApp

Global.asax.vr

```
//4.0.5
BegSr Application_Start
      DclSrParm sender Type(*Object)
      DclSrParm e Type(EventArgs)
   DclFld ActiveJobTable Type(System.Collections.Hashtable)
   Application.Lock()
   ActiveJobTable = *new System.Collections.Hashtable()
   Application["ActiveJobs"] = ActiveJobTable
   Application.UnLock()
EndSr

BegSr Session_Start
      DclSrParm sender Type(*Object)
      DclSrParm e Type(EventArgs)

   DclFld Device Type(Object)
   DclFld Job Type(ASNA.Monarch.WebJob)
   DclFld ActiveJobTable Type(System.Collections.Hashtable)
   DclFld fileName *String

   fileName = System.IO.Path.GetFileNameWithoutExtension(Request.Path)
   If (fileName.StartsWith("!") *Or Request.Form["__isDspF__"] <> *Nothing)
         LeaveSR
   EndIf
   Session("MonarchInitiated")  = *Nothing
   Job = NewJob()
   Session("Job") = Job
   Device = Job.Start(*this.Session.SessionID)
   Session("Device") = Device
   ActiveJobTable = Application["ActiveJobs"] *As System.Collections.Hashtable
   EnterLock (ActiveJobTable.SyncRoot)
        ActiveJobTable.Add (*this.Session.SessionID+Job.PsdsJobNumber.ToString(),
Job)
   ExitLock
EndSr

BegSr Session_End
     DclSrParm sender Type(*Object)
     DclSrParm e Type(EventArgs)
     DclFld ActiveJobTable Type(System.Collections.Hashtable)
     DclFld Job Type(ASNA.Monarch.WebJob)
```

```
    Job = Session("Job") *As ASNA.Monarch.WebJob
    If (Job <> *Nothing)
        Job.RequestShutdown(20)
    EndIf

    ActiveJobTable = Application["ActiveJobs"] *as System.Collections.Hashtable
    EnterLock (ActiveJobTable.SyncRoot)
        ActiveJobTable.Remove(*this.Session.SessionID+Job.PsdsJobNumber.ToString())
    ExitLock
 EndSr
 BegFunc NewJob Type(ASNA.Monarch.WebJob)
  DclFld  JobsAssembly Type(System.Reflection.Assembly)
  JobsAssembly = System.Reflection.Assembly.Load("MyCompany.MyApplication.MNCHCL")
  LeaveSR JobsAssembly.CreateInstance("MyCompany.MyApplication.MNCHCL_Job.MyJob")
*as ASNA.Monarch.WebJob
 EndFunc
```

MyJob.vr

```
//4.0.5
Using System
Using MyCompany.MyApplication
DclNamespace MyCompany.MyApplication.MNCHCL_Job

BegClass MyJob Extends(ASNA.Monarch.WebJob) Access(*Public)
   DclDB Name(MyDatabase) DBName("*Public/CHERRY") Access(*Public)
   DclDB Name(MyPrinterDB) DBName("monarchtargetDB") Access(*Public)

   BegFunc getDatabase Type(ASNA.VisualRPG.Runtime.Database) Access(*Protected)
                       Modifier(*Overrides)
      LeaveSR MyDatabase
   EndFunc

   BegSr Dispose Access(*Public) Modifier(*Overrides)
      DclSrParm disposing Type(*Boolean)
      If disposing
          Disconnect  MyDatabase
      EndIf
      *Base.Dispose(Disposing)
   EndSr

   BegSr  ExecuteStartupProgram Access(*Protected) Modifier(*Overrides)
      Connect     MyDatabase
      StartTPM Db(MyDatabase) Level(*Low)
      CallD        'MyCompany.MyApplication.Mnchcl'
/Error This entry-point requires parameters. Please add the parameters to be used in
this call. (You may use F12 with the cursor on the program's name to jump to the
program's definition).
      EndTPM Db(MyDatabase)
EndSr
EndClass
```

Non-Interactive

Console Application

A console application is a Process application that is user-initiated - such as selecting a menu option. It is an entry point into a Subsystem. If it receives any parameters, they are typically input-only. It does not return any values.

Feature	Value
Entry points	1
Display Files	No
Caller	User
Started	Command Line or OSEXEC

Implementation

Visual Studio Project Type: **Console Application** (no printer).

MyJob.vr

```
// 4.0.5
Using System
Using MyCompany.MyApplication
DclNamespace MyCompany.MyApplication.MNCHCL3ca_Job


BegClass MyJob Extends(ASNA.Monarch.Job) Access(*Public)
   DclDB Name(MyDatabase) DBName("*Public/CHERRY") Access(*Public)

   BegFunc getDatabase Type(ASNA.VisualRPG.Runtime.Database +
                    Access(*Protected) Modifier(*Overrides)
        LeaveSR MyDatabase
   EndFunc
   BegSr Dispose Access(*Public) Modifier(*Overrides)
     DclSrParm disposing Type(*Boolean)
     If disposing
        Disconnect  MyDatabase
     EndIf
     *Base.Dispose(Disposing)
   EndSr

   BegSr ExecuteStartupProgram Access(*Protected)
      Connect      MyDatabase
      CallD        'MyCompany.MyApplication.Mnchcl3'
      DclParm _X Type( *Packed )      Len(10, 0) // added
      EndPrograms()
   EndSr

   BegSr Main Shared(*Yes) Access(*Public)
     DclSrParm Args Type(*String) Rank(1)
     DclFld job Type(MyJob) New()
     job.ExecuteStartupProgram()
  EndSr
EndClass
```

Class Library

A class library is a collection of Programs, none of which uses Display files. One or more are entry points called by other Programs outside of the class Library. The entry points typically

have input and output parameters. This class Library is similar to the ILE service Program concept.

Feature	Value
Entry points	1 or More
Display Files	No
Caller	Other Programs
Started	Call/Parm

Implementation

Visual Studio Project Type: **Class Library**

MyJob.vr

```
// 4.0.5
Using System
DclNamespace MyCompany.MyApplication.MCCHCL3cl_Job

BegClass MyJob Extends(ASNA.Monarch.Job) Access(*Public)
   DclDB Name(MyDatabase) DBName("*Public/CHERRY") Access(*Public)
   BegFunc getDatabase Type(ASNA.VisualRPG.Runtime.Database) +
           Access(*Protected) Modifier(*Overrides)
       LeaveSR MyDatabase
   EndFunc
EndClass
```

Creating GamePlans

A **GamePlan** is a subset of a Gallery. To create a GamePlan you specify an entry point - a **program**, **service program**, **menu**, or **database file** – then Cocoon includes all objects this entry point references. Other references outside the scope of the GamePlan entry point are included as 'external' references. You can add additional objects after creating the GamePlan.

Before creating your GamePlan, it is important to establish directives that become a subset of the directives for the GamePlan. It is easier to establish these beforehand than to try to change each individual directive after creating the GamePlan. As an object is added, the object directives established in the Gallery are copied as the initial directives into the GamePlan.

There are several access methods (menus) available within Cocoon when you are ready to create a GamePlan. These menu options can differ but after selection, each requires the entry of a GamePlan name and the database name to hold the new GamePlan (see **Figure 1**).

When a specific object is not pre-selected as an entry point before the option to create a GamePlan is chosen, you will need to specify the entry point and library list on the Definition tab (see **Figure 2**).

The following briefly touches on the different entry point object types when creating a GamePlan.

Creating a GamePlan from a Program or Service Program

Select a **Program or Service Program** from the Section Call Graph, Cluster Call Graph, or the solution explorer window. All objects referenced by the entry point will be included in the GamePlan.

Creating a GamePlan from a Cluster

Select a **Program Cluster** from the Section Call Graph, Cluster Call Graph, or the solution explorer window. All objects referenced by the entry point and all programs in the cluster will be included in the GamePlan.

When multiply entry points are present, a single entry point is chosen arbitrarily. You can use the **Set as Entry Point** option after creating the GamePlan or modify the program name in MyJob.vr created during migration.

Creating a GamePlan from a Menu

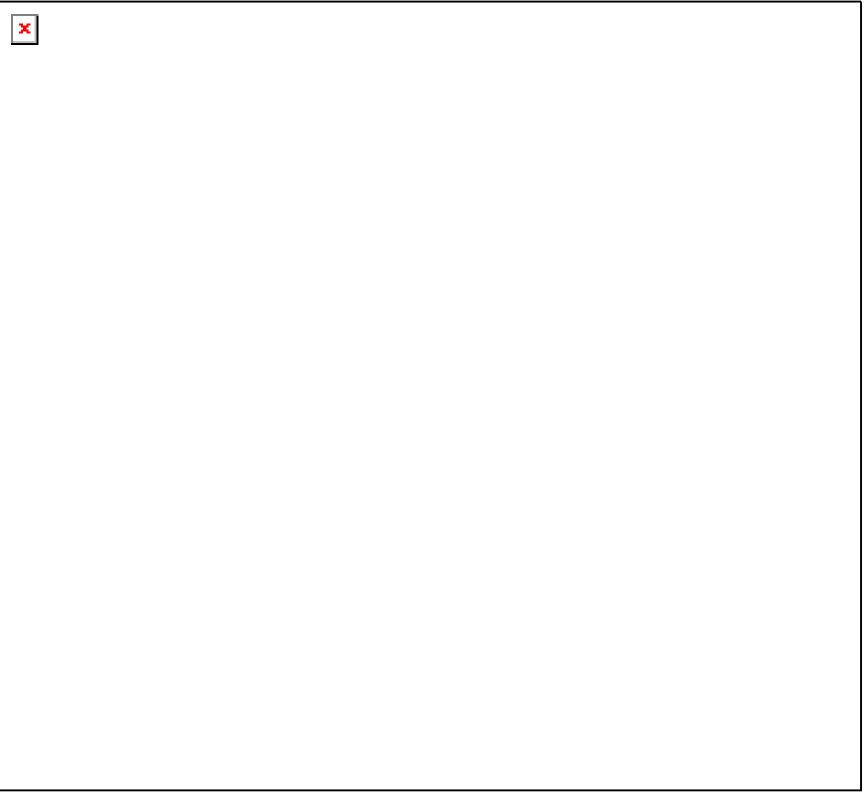
Select a **Menu** from the solution explorer window. All objects referenced by the menu, and all object they refer too, will be included in the GamePlan.

Creating a GamePlan from a Database

Select the **Database file** from the DB Files Graph. All files referenced by the file selected will be included in the GamePlan.

Figure 1

Figure 2

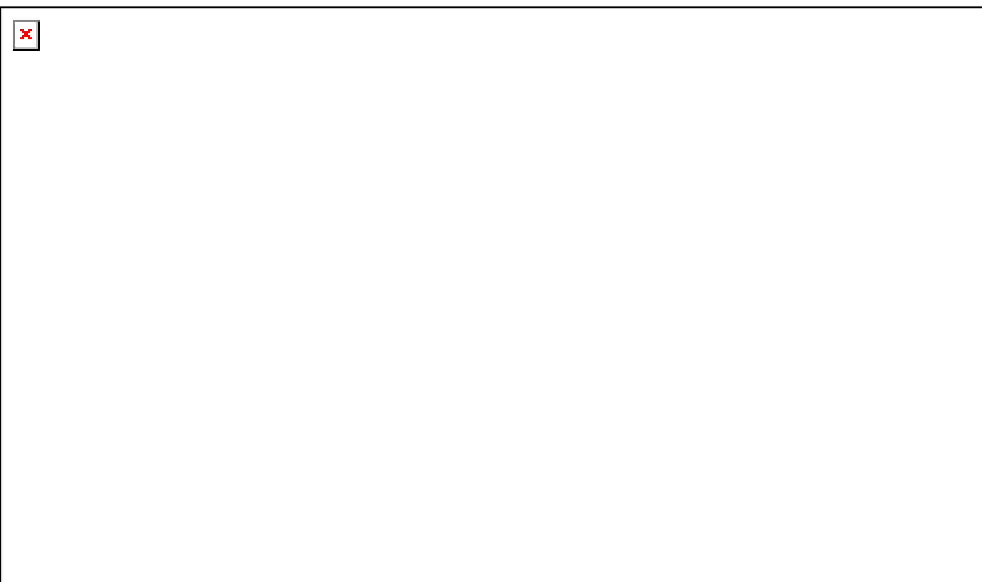


Including Additional Objects

After a GamePlan is created, there may be instances when it is advantageous to add additional objects to the GamePlan manually. One case might be Message Files, which cannot be identified as being used by a particular program.

After selecting the **Add Object** menu option, you can select objects to be added to the GamePlan as shown in the figure below.

> ***Note:*** *Only those objects* ***not*** *already included are shown under the* ***Objects in Gallery and not in GamePlan*** *column. To view* ***all*** *the objects in the Gallery, check the* ***Show All Objects*** *checkbox.*

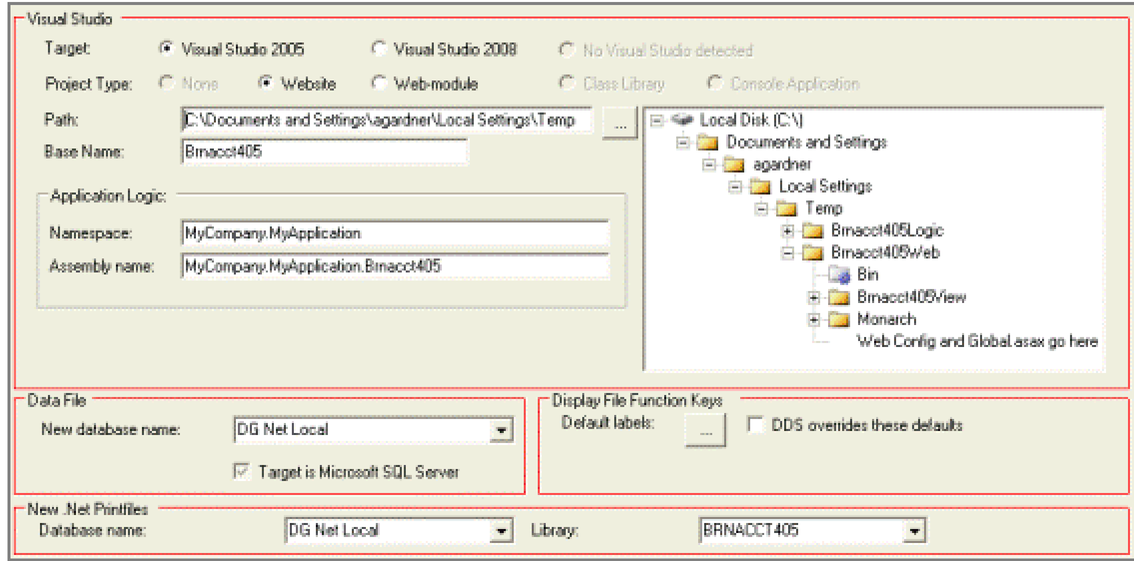


Establish GamePlan Directives

One of the first duties after creating the GamePlan is to establish the GamePlan **Directives**. See **Chapter 12 – Migrating Data** for information on directives for a Data only GamePlan.

The top of the Directives page shows the GamePlan *Name*, *Gallery*, and *Entry Point*.

Each outlined section is detailed below.



Visual Studio Options

The next section of the GamePlan Directives deals with the type of Visual Studio Project that will be generated. As discussed previously, the choices are:

1. None
2. Website (ASP.NET Web Class Library)
3. Web-module
4. Class Library
5. Console Application

If there is at least one display file in the GamePlan, then Cocoon allows only the Website or Web-module Project types. Conversely, if there are no Display Files included in the GamePlan, you can only choose between None, Console Application, and Class Library.

Depending on the type of project, the information required will be different. For instance, if Project Type is **None**, there are no Visual Studio options since a solution is not created.

Setting the ***Visual Studio Target*** allows the solution to be created for Visual Studio 2005, Visual Studio 2008, or neither depending on what is detected by Cocoon. If VS 2008 is detected, you can specify the *Framework* version as .NET Framework 2.0, .NET Framework 3.0, or .NET Framework 3.5.

For a **Website** you enter the *Path*, for a **Web-module** you enter the *Host Solution;* both require a *BaseName*. The *BaseName* is the variable in the folder structure displayed in the **Preview** window. The Application Logic *Namespace* and *Assembly name* can be entered but both will default to directive values previously established when the Gallery was initially created (Chapter 2 - Creating a Gallery).

For a **Class Library** and **Console Application** project, the Visual Studio section of the directives is different. You enter a solution *Name*, *Namespace*, *Assembly Name*, and *Project locations* (as shown below).

Data File

This section deals with a DataGate database name pointing to where the actual data can be found.

The **New database name** is used to establish the connection to the database server when the newly generated applications is compiled and run. The name is used in the AVR DclDB command generated in the MyJob class. This database is available as MyJob.MyDatabase at compile time or as monarchJob.Database at run-time. It is also used in the DclDiskFile generated commands, in the remote Program calls, and in the DataArea IN and OUT operations. Library list settings in the database named should be established such that all of these objects can be found.

Display File Function Keys

Default Labels

When a Display File is migrated, the Function Keys available at the File or Record level are converted into buttons. The availability of function keys is determined by the use of the DDS Keywords CFxx or CAxx (where xx = 1-24) and other keywords like PRINT or PAGEUP. These keywords have an optional comment. This comment can be used as the generated button's label. There are many instances where there is no comment provided in the DDS source so there is no way of determining what a good label would be for the buttons, then the function key name is used.

The **Default Labels** entry allows you to change the default function key labels for all the Display Files to be migrated. The **DDS Override** checkbox allows you to override the usage of the DDS command as labels for function keys.

New .NET Printfiles

Similar to the database name used for data files, a DataGate **Database name** is needed to locate the migrated print files.

Migrating Print files involves reading DDS or O-Specs and creating DataGate print files. Since it might be convenient to store the new Print files back to the original Legacy server, enter a *Library* name to avoid creating a DataGate print file in the same Library where the legacy OS/400 print file exists. **Both kinds of Print files cannot exist in the same server under the same Library.**

This database is also used to generate a DclDB in the MyJob class, which is referred to in the DclPrintFile-generated commands.

Checking the Task List

Once the Directives are set on a GamePlan, the next step is to run the migration agents to report on any tasks found that may require Migrator intervention.

The list of tasks is obtained by invoking the corresponding migration agent for each object in the GamePlan. The agent analyzes the object source and usage and emits a list of issues that have to be addressed manually.

Using Filters

Notice the drop-down boxes on the top of the Task List as well as the *Base Filter* button and *Use Base Filter* checkbox.

The first set of simple filters allows you to filter not only the contents of the Task List, but the Summary and Effort panels also. These filters are always active and can enhance the base filters when they are used.

First Filters

Each drop-down box will only include those selections valid for the task list. The *Category* selection includes problems, unsupported, and exceptions. *Task ID* allows selection of a

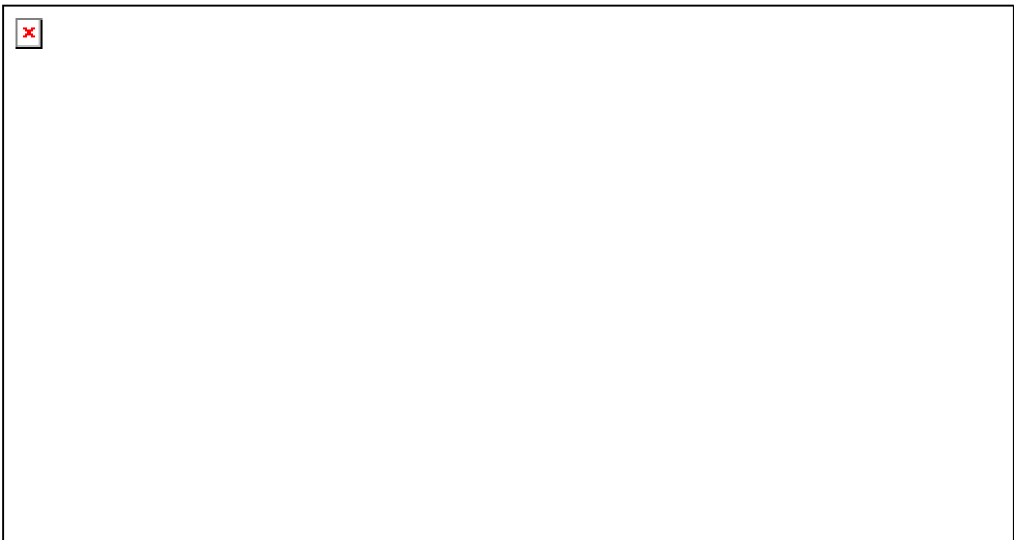
specific task. *Severity* is an indication, 10 to 40 (warning to must fix), of how serious a task is. *Object Type* shows the types of objects shown in the list. *Agent* is the selection from the agents producing the task.

Base Filters

The base filters are subsequent to the first filters. Once the base filters are entered, the rules are retained (albeit ignored) if the checkbox *Use Base Filter* is unchecked. This makes it easy to switch the base filters on and off without the need to re-enter the rules statements.

To activate (or enter) base filters, check the *Use Base Filter* checkbox and then select the *Base Filter* button. The **Edit Base Filter** dialog box displays to allow the rules to be entered in the fixed format grid or the free format Advanced tab, which is used to enter a .NET expression as the selection statement.

The following image shows an Edit Base Filter dialog box with the Grid filled in to select tasks generated by the CL Agent, RPG Agent, or the Display File Agent, with a Severity equal 30.



Filter Selection Rules

Consider this when you establish your filters.

a) *The 'First' filter fields are all contained in a single condition comparison IF statement; a task evaluated true is selected, otherwise it is omitted.*

b) *When a task is omitted by the First filters, any 'Base' filter rule statements are ignored.*

c) *Only after a task is selected by the First filters are the Base filter rule statements applied to that task.*

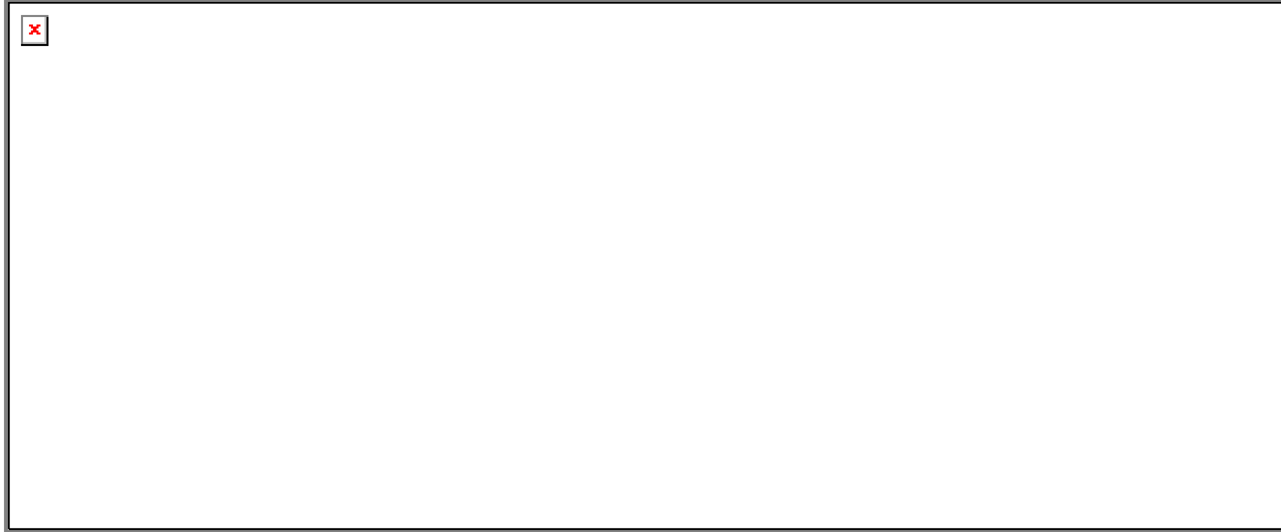
d) *Base file rules can be compound if and/or statements.*

For instance, if you want to view tasks for both the CL Agent and the RPG Agent, enter **both** rule statements as Base Filters AND enter "All" in the First filter *Agent* field. If you were to

enter "CL Agent "in the First filter *Agent* field, any "RPG Agent" tasks would be omitted by that first rule and ignored.

Summary Panel

The Task List SUMMARY panel provides a total for each task ID yet you can still expand a specific ID (1023 shown below) to view the detail.



Categories

The tasks are organized in the following categories:

- Problems
- Unsupported Features
- Exceptions

Problems

This category of tasks involves significant issues that typically prevent the agent from doing a thorough analysis of the corresponding object. Examples of this category of tasks are:

- Failed to find Visual Studio directory
- Could not connect to server
- Project location is not valid
- Object referenced by Program was not found
- Web Site Name is not specified
- Copybook was not found
- Unauthorized Access while attempting to write
- Some O-Spec records fetch for Overflow, but no fields are conditioned with OF indicator
- Printer device specified, but O-Specs have not been discovered
- Call to Program with missing Directives data
- Failed to locate legacy source for print file
- Net Print file already exists in Database/Library
- Field Reference File was not found

You should try to fix all listed problems before proceeding into the actual migration.

Unsupported Features

The **Unsupported Features** category groups all those features of OS/400 constructs that are not supported by Monarch. These tasks point to areas where the Migrator will have to create alternative solutions. Examples of the kinds of tasks in this category are:

- Unsupported CL Command and CL keywords
- Unsupported: Parameters used with keyword
- File is Not an Externally Described File
- File is Not a full procedural file
- Field not supported for Program Status DS
- Unsupported DDS keyword

Exceptions

Finally, these are the tasks issued when Cocoon or one of its agents find an unexpected situation that cannot be handled (read as bug).

> ***These tasks should be reported to ASNA.***

Sizing Migration

Monarch can assist in measuring a migration job. There are two ways of viewing the size of a migration job. The first entails obtaining raw metrics on the original application and the second focuses on the effort required to manually resolve the unsupported features. Monarch refers to the first as **Magnitude Density** and the second as **Effort Density**.

Monarch provides mechanisms for the Migrator to influence the sizing of a GamePlan by refining the parameters used in the measuring algorithm.

Magnitude Density

The Magnitude Density provides a measure of the size of the GamePlan, based mostly in lines of code, but it also takes into account other attributes of the different sources. Only these objects that have source lines **and** are marked for migration are taken into account while computing the magnitude.

- CL Program
- RPG/400 Program
- RPG ILE Program
- Display File
- Printer File

Algorithm:

For each measured object, the number of lines in its source is multiplied by a 'line cost'. By convention an RPG line of code counts as one, all others should be relative to this one, the line cost factoring attempts to make DDS or CL lines comparable to RPG. While this is not exact, it permits combining an application's entire source into a single number. The default costs of each object type are as follows:

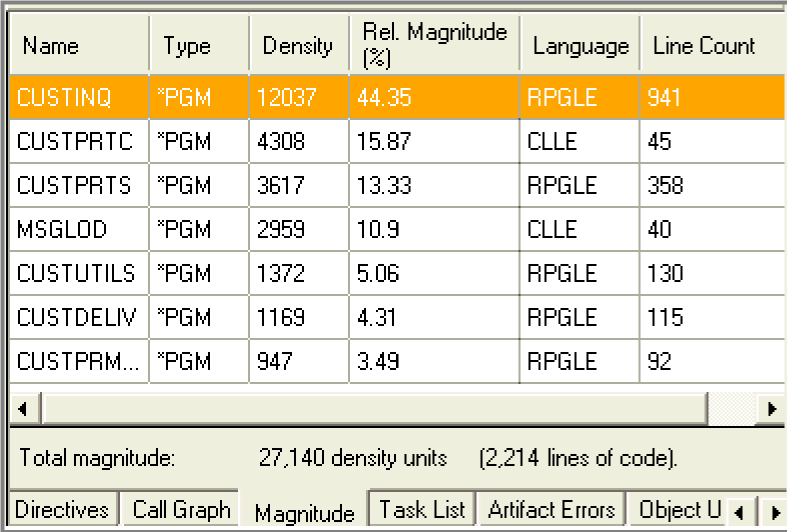
Object Type	Line Cost
CL Program	5
RPG/400 Program	1
RPG ILE Program	1
Display File	10
Print File	10

The density for Display and Print files is the DDS line count multiplied by the Line Cost.

For programs, in addition to pure lines of code, certain other program attributes are included in the computation of a program's magnitude density. Each one of these attributes is multiplied by a weight factor to compute an object's density. The following table shows the default weight for each attribute:

Attribute	Weight
Line Count * Line Cost	1.0
Number of parameters	10.0
Called Programs	2.0
Called Parameters	5.0
Called 3rd Party or unknown programs	4.0
Database Files Used	1.0
Workstation Files Used	5.0
Print File Used	4.0
Data Areas Used	2.0
Programs Calling this one	1.0

If any error is encountered in the process of computing an object's density, the resulting line is marked with a different color. The individual magnitude density is best utilized when comparing amongst the densities of other objects in the GamePlan. Cocoon shows the density totals in a tab associated with the GamePlan. The following figure shows an example.



Name	Type	Density	Rel. Magnitude (%)	Language	Line Count
CUSTINQ	*PGM	12037	44.35	RPGLE	941
CUSTPRTC	*PGM	4308	15.87	CLLE	45
CUSTPRTS	*PGM	3617	13.33	RPGLE	358
MSGLOD	*PGM	2959	10.9	CLLE	40
CUSTUTILS	*PGM	1372	5.06	RPGLE	130
CUSTDELIV	*PGM	1169	4.31	RPGLE	115
CUSTPRM...	*PGM	947	3.49	RPGLE	92

Total magnitude: 27,140 density units (2,214 lines of code).

Directives | Call Graph | Magnitude | Task List | Artifact Errors | Object U

Customizations

A Migrator can customize the weights and costs involved in the computation of the Magnitude Density by modifying the ***cocoon-preferences.xml*** file found typically \Documents and Settings\<userName>\Local Settings\Application Data\ASNA\Cocoon. The file includes a section named ***<DensityCalculator>***, shown below, where the default weight for each parameters can be modified.

Effort Density

While having an idea of the metrics involving lines of code in a GamePlan is useful in sizing a migration job, it is not enough to have a clear idea of the complete effort required in doing such migration. The Effort Density assigns a number to effort required to migrate individual objects based on the number and type of tasks discovered by the migration agents. These tasks are typically unsupported features, but some of them involve problems with the original applications, mostly with the location and authenticity of the sources. Only those objects that are marked for migration are taken into account.

Algorithm

Given a Task List, a number is computed for each object representing the effort required to migrate it.

Each task has a number assigned representing the cost (expressed in minutes) associated in resolving the task. The Effort Density of an object is the sum of all the costs associated with each task found for the object. The following figure shows an example Effort Density.

Each task has its own cost assigned to provide a more accurate measurement. Here is an example of tasks and their corresponding costs:

Task	**Description**	**Cost**
3004	Monitor for CPF0000 needs manual conversion to Try ... Catch	10
4034	Error while writing Copybook {0} Reason : {1}	240
4037	Unsupported: Data structure field '{0}'; initialized by {1}	15

6044	Reference Format '{0}' for File '{1}' not found	240

Customizations

Migrators with different levels of knowledge and experience might take different amounts of time to resolve particular types of tasks. The cost associated with each task can be fine-tuned to the individual Migrator in cocoon's message file ***Cocoon.Migration.MsgFile*** found in the same folder where cocoon.exe is found, typically /Program Files/ASNA/Monarch 4.0/Cocoon.

Chapter 5 - Pulling the (Migration) Trigger

Executing the migration process is a simple one-click operation. However, the effort for the Migrator is not in selecting a menu option, but rather preparing the GamePlan by establishing Directives, securing source files and their locations, and removing all problem tasks reported in the Task List as explained in the previous chapters.

When the Migrate menu option is invoked, Cocoon goes to work by inspecting the Directives on each object and invoking the corresponding migration agent.

Currently, there are 8 migration agents and one Solution Builder. The following chapters will provide more information on each one of these agents. A future release will fully introduce the Menu Agent.

Several of these agents operate with objects outside the GamePlan itself, namely templates.

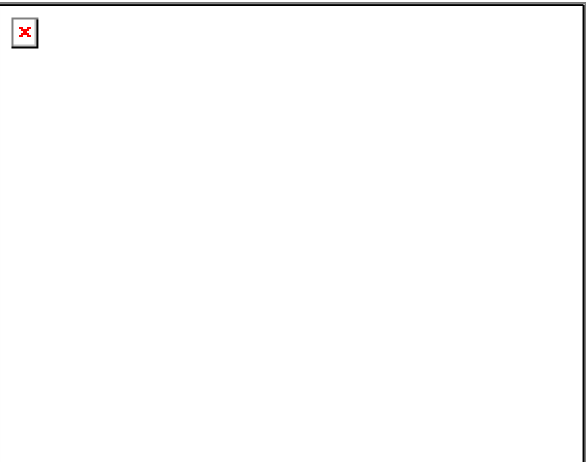
Establishing Templates and Cascading Style Sheets

Except for Data Only GamePlans, it is important to establish the templates and cascading style sheets for the agents before starting the migration process. These templates influence the output of the migration agents. Some templates include a reference to a cascading style sheet (css) that determines the look of the generated page. If a change were introduced later on in an agent's template or cascading style sheet, the objects affected by that agent would have to be re-migrated to include these changes.

It is important to know that changing a template does not affect previously migrated objects. Take for instance, the display file agent. The template and cascading style sheets controls the look of the ASPX pages generated. It is common to modify individual pages or the style sheets after they have been migrated.

If you want a particular common format for all pages, then establish the template and cascading style sheets **before** migrating. Otherwise, you will need to be run the migration process again as the Display Files and the individual modifications previously made to the pages would be lost.

The default templates files are located in the installation default folder as shown:



The following figure shows the template files within the Templates\DisplayFiles sub-folder.

The risk of modifying the files in the **Templates** sub-folders is that these files are replaced when you re-install Monarch. The solution is to create you own **Templates** folder and place **ONLY** the files you have modified into the appropriate sub-folder.

> ***Note:*** *Any templates or style sheets you create must be placed in the appropriate sub-folder that parallels the original installation of "Templates". Then you need to set the* ***Agent's template root path*** *to direct the migration agents to this new templates location (found on the* ***General Default Directives****).*

The migration Agents will attempt to acquire needed templates and cascading style sheets from this template path **first**. If **not found**, then Monarch uses its original default location to locate templates and cascading style sheets.

Cascading Style Sheets

There are two main and seven secondary default cascading style sheets in Monarch Cocoon located in C:\..\Templates\DisplayFiles as of this writing.

The two main css files import all of the secondary default files. If any of the secondary file names change, you must change the @import references in this two main file (and include them in C:\MyTemplates\DisplayFiles). The two main css files are:

- **Monarch.css** - Imports all of the secondary css files and otherwise controls the location of form elements on the rendered page allowing for the inclusion of a logo at the top (default is Monarch_logo.jpg).
- **WinPopUp.css** - Imports all of the secondary css files and otherwise controls the location of form elements on the rendered page without a logo at the top.

The seven secondary css files are provided to allow for each type of control to be separated for ease of maintenance but styles can be added to any of these files as best fits your needs. These files contain the styles used in the CssClass and CssEnterClass properties that control the appearance (font, font weight, point size, underline, italics, color, etc.) of individual controls. During migration, the Solution Builder in ASNA Monarch Cocoon applies the CssClass property from the styles defined in these files.

Refer to each file for the styles present in each of the secondary files:

- Common.css
- DdsCharField_Control.css
- DdsConstant_Control.css
- DdsDateField_Control.css
- DdsDecField_Control.css
- DdsFile_Control.css
- DdsRecord_Control.css

Display File Templates

There are two aspx files using these cascading style sheets. They are:

- **DisplayFiles\Template.aspx**: This file includes a link reference to **Monarch.css** and includes DDS controls for placeholders to indicate the location of the generated DdsFile and DdsRecord in the page. See **Migrating Display Files** in Chapter 6 for more detailed information.
- **AspWebApp\WinPopUp.aspx:** This file includes a link reference to **WinPopUp.css** and includes one placeholder for popup windows. See **Solution Builder** later in this chapter for more detailed information.

Note: *Before making any of the changes referred to below, it is assumed that you have copied the files you want to change to the appropriate sub-folder and that any changes made are to the copied files; not the originals.*

These two files are mentioned specifically because of their reference to the cascading style sheets where some general rules apply.

- If you change any cascading style sheet **without** changing the name of the file, only the changed .css files need to be in your new templates location.
- If you change either Monarch.css or WinPopUp.css **AND** the file names changed, you must also change the link reference in Template.aspx and/or WinPopUp.aspx to reflect the new .css file name.
- If you change Common.css **AND** change the name of the file, you must change the @import statement in **both** Monarch.css and WinPopUp.css to reflect the new css name.

Subsequent sections or chapters in this guide deal with the individual agents in more detail. Additional information will be provided, as appropriate, for templates that you may need to review for modifications.

> ***However**, always refer to "C:\Program Files\ASNA\Monarch 4.0\Cocoon\Templates" for the current sub-folder structure, the sub-folder files, and the **content** of each file to determine if changes are needed.*

Master Pages Considerations

Consider using ASP.NET Master Pages to allow you to create a consistent layout for all the pages in your application that define a "look" and standard behavior. In addition to the Master Page, individual Content Pages are used to contain just the content you want to display. When the Content Page is requested, it is merged with the Master Page combining the layout of the Master Page with the content from the Content Page. Refer to Visual Studio topic **ASP.NET Master Pages Overview** to review the concepts of Master Pages to see if they are right for your application.

Templates\WebRoot Considerations

This special folder is provided as a means to copy its content directly into the web root of your Visual Studio solutions. This folder may contain master pages, images, cascading style sheets, site map pages, etc.

The WebRoot folder is set up similar to templates in that you create a new WebRoot folder paralleling the originally installed WebRoot and set the ***Agent's template root path*** in the migration directive to your templates location (*C:\MyTemplates* in the example above).

> ***However, there is one major exception. The installed WebRoot folder and your new WebRoot folder are mutually exclusive****.*

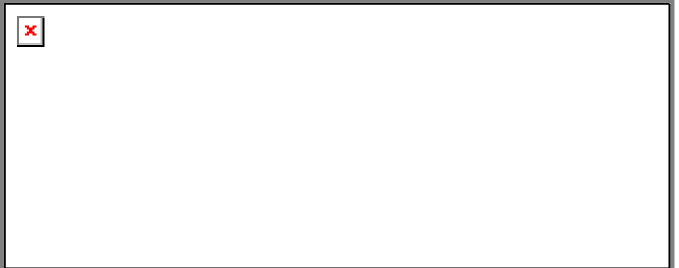
If there is a WebRoot folder in the ***Agent's templates root path***, the **entire contents** are copied to the web root of your solution; otherwise, the WebRoot installation folder contents are copied. **The contents of the two folders are NOT combined.**

If you want to use the installed files provided but have additional files or a file you want to change or replace, follow these basic steps:

- Copy the entire contents of the installed WebRoot folder to the agent's templates root path location
- Add any additional files
- Make changes to or replace existing files
- Set the *Agent's templates root path* to the templates location

Example:

The following figure shows some files within MyTemplates\WebRoot that have been established that will be copied into the web root of the Visual Studio solution (provided Agent's templates root path is set to C:\MyTemplates).



Solution Builder

The Solution Builder generates the Visual Studio Solution and its Project files and folders. It is also responsible for generating the **MyJob** class.

The Solution Builder uses the templates shown below (located under the ***Templates*** folder).

Project Type	Templates Folder	Contents
Website (ASP.NET Web)	AspWebApp	!Diagnose.aspx !Diagnose.aspx.vr !EoJ.aspx !EoJ.aspx.vr !ExpiredSession.htm AppSettings.config Global.asax.vr MyJob.vr User.js WinPopUp.aspx WinPopUp.aspx.vr
Class Library	ClassLibrary	MyJob.vr
Console Application	ConsoleApp	MyJob.vr

Templates and Cascading Style Sheet Considerations

You may consider changing the content of the following files used by Solution Builder during migration:

- !Diagnose.aspx - Used to display the messages for the application end-point for crashed sessions.
- !EoJ.aspx - Used as an application End of Job page. You can use !Eoj to redirect the user to the main menu or home page.
- !ExpiredSession.htm - Used as an application end-point for expired sessions. !ExpiredSession is **not** an ASPX page to avoid starting a new session.

Changes to the following files may also be considered as needed.

- WinPopUp.aspx (which contains a link reference to WinPopUp.css)
- DisplayFiles\WinPopUp.css (which contain an @import of all the remaining files listed)
- DisplayFiles\Common.css
- DisplayFiles\DdsCharField_Control.css
- DisplayFiles\DdsConstant_Control.css
- DisplayFiles\DdsDateField_Control.css
- DisplayFiles\DdsDecField_Control.css
- DisplayFiles\DdsFile_Control.css
- DisplayFiles\DdsRecord_Control.css

Other Considerations

The file **User.js** is an empty JavaScript file provided so that you may include any java script that may be needed for your unique application. During migration, a reference to this file is automatically added to the code behind in every aspx page in the website as:

```
<script language="javascript" type="text/javascript" src="../Monarch/User.js" ></script>
```

This makes it easy to add java script to the User.js file since it is already referenced in your aspx pages.

Chapter 6 - Migrating Display Files

Display Files Overview

An OS/400 Display File is composed of one or more formats, each defining a template of what is to be shown in a segment of the screen. Each format may contain input and output fields and constants as well as directives on how to display each of these elements and the interaction of this format with other formats present on the screen.

The Display File Agent creates an ASP.NET form taking as input the OS/400 display file's DDS specifications and a user-provided template that are migrated into Active Server Pages for .NET forms. The migration strategy for each display file is to create an ASP.NET form composed of two files.

First, there is an '.aspx' file containing HTML, ASP, and Monarch elements describing the layout and data composition of the display using server controls provided by Monarch.

Second, is the '.aspx.vr' file containing the code-behind class that extends the ASNA.Monarch.WebDspF.Page class. This class is part of a framework providing the run-time support to make the display file appear to the user.

> ***Note**: To re-migrate a display file, first delete both the .aspx and .aspx.vr for the file in Visual Studio.*

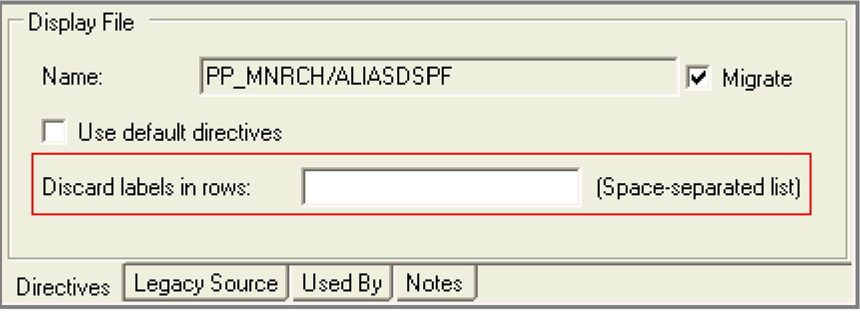
Most of the elements found in the DDS specification are migrated. However, two general features are ***not*** dealt with - **Help and window widgets**. The help system defined in the DDS is alien to the web model and it is not migrated. In the mid 90s, DDS incorporated a set of keywords to 'facilitate' the creation of Windows-looking screens. Because specialized hardware was needed to render these keywords properly, the practice of using them never caught on. The one big exception is the support for the WINDOW keyword that is converted to a browser pop-up window.

The following figure shows a diagram of the Display File agent.

Display File Directives

These directives control function key labels for DDS Display Files. When *Use default directives* is checked, the default preferences established in Chapter 2 are used.

When *Use default directives* is unchecked (as shown below), *Discard labels in rows* appears. These entries are used to throw away any redundant labels in old screens that were used to display the function key command description. The migrated aspx will show buttons for active function keys with nice labels (and tool tips) making the old labels redundant.



Templates

The Display File Agent uses a template to determine the look of the generated page. The templates are located in the *Agent's templates root path or* the installation *Templates* folder.

A template is a 'regular' aspx page pair (.aspx and the .aspx.vr) with requirements as follows.

Template.aspx

The Template.aspx file must include a single DdsFile and one DdsRecord. These controls are used as placeholders to indicate the location of the generated DdsFile and DdsRecord (including subfiles) in the page. Certain properties of the placeholder controls are copied to the generated controls. The DdsRecord must be a direct child of a control imposing a **flow** layout (as opposed to grid layout).

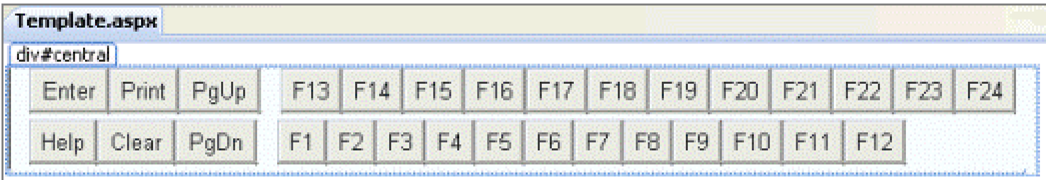
This aspx includes a reference to Monarch.css mentioned earlier.

Template.aspx.vr

The Template.aspx.vr file contains a class derived from **ASNA.Monarch.WebDspF.Page**. The source may contain declaration for DDS widgets but they are removed from the final generated source.

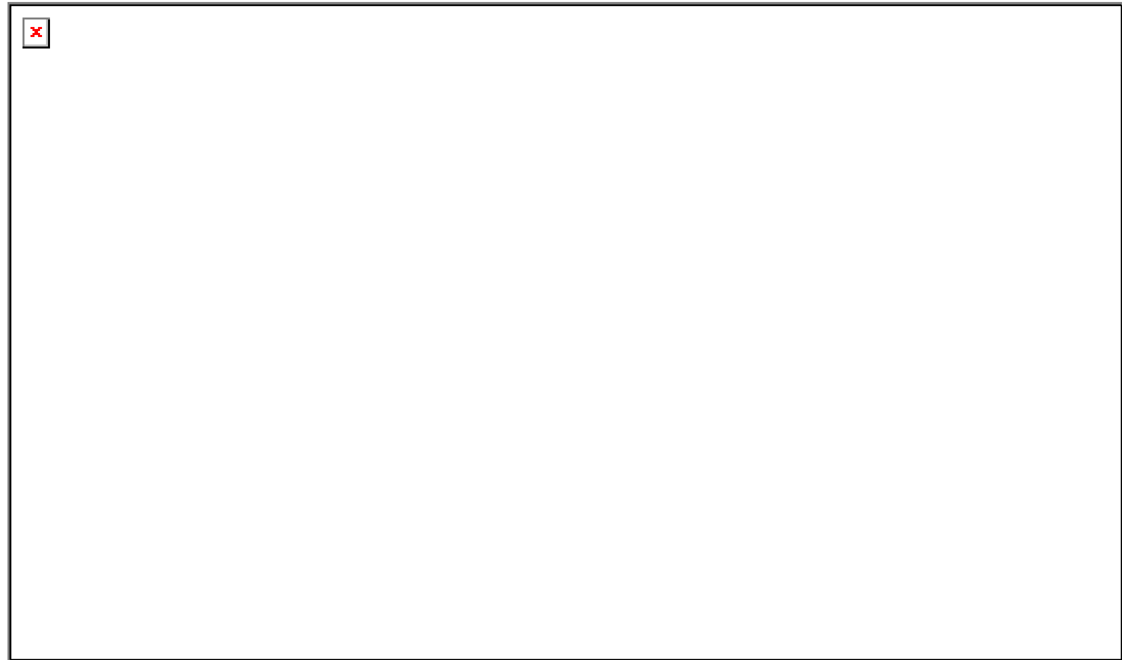
Example

The following is an example of a display file Template.aspx (VS2008) and Template.aspx.vr.



Template.aspx Code

In the following image, the @Register and <mdf:ddsrecord> have been separated into two lines for readability but will be one line in your solution.



Template.aspx.vr Code

```
DclNamespace {0}
Using System
Using System.Data
Using System.Web
Using System.Web.SessionState
Using System.Web.UI
Using System.Web.UI.WebControls
Using System.Web.UI.HtmlControls
BegClass {0} Partial (*Yes) Extends(ASNA.Monarch.WebDspF.Page)
    …
EndClass
```

This page is intentionally left blank.

Chapter 7 - Migrating CL Programs

A CL Program on OS/400 has the capability of using over a thousand commands plus any number of user-created commands. Usage of CL can be divided into two major categories: **System Administration** and **Application Coordination**.

The administration of the system involves operations like the creation of user profiles and the save/restore procedures. These activities are not considered part of the application itself and Monarch does not attempt to facilitate such activities. Normal Windows techniques should be employed to affect these kinds of activities.

On its other personality, CL is used to setup the environment for RPG Programs to run. Typical functions done by CL in this context are:

- Set up the Library List
- Override database file
    - Select a different File or Member to be used by the RPG 'F' spec
    - Establish a Query File
- Maintain application parameters in Data Areas
- Allocate objects
- Control the Program Message Queue

These actions affect the OS/400 Job where Programs are running and have an effect on all Programs in the same job. Monarch Framework provides these facilities through a set of classes that supplement the .NET Framework.

The following list shows the CL commands that are supported by Monarch.

- PGM, ENDPGM
- IF , ELSE, DO, ENDDO , GOTO
- DCL, CHGVAR
- CALL, CALLB, RETURN
- MONMSG
- ADDLIBLE, RMVLIBLE
- DCLF, RCVF, SNDF, SNDRCVF
- OVRDBF, OVRDSPF, DLTOVR
- CLRPFM, INZMBR
- CHGDTAARA, RTVDTAARA
- RTVJOBA
- SNDPGMMSG, RMVMSG
- CRTDUPOBJ (for files only)

Resources

Alphabetic List of CL Commands:
http://publib.boulder.ibm.com/iseries/v5r2/ic2924/info/rbam6/rbam6alphatable.htm

CL **Command Finder** page
http://publib.boulder.ibm.com/iseries/v5r2/ic2924/info/rbam6/rbam6clfinder.htm.

See the following resource for more information about using CL. **CL Programming** (http://publib.boulder.ibm.com/iseries/v5r2/ic2924/books/c4157215.pdf). This book provides a wide-range discussion of Programming topics, including control language Programming, Programming concepts, objects and libraries, message handling, and user-defined commands and menus.

Chapter 8 - Migrating RPG Programs

Program Migration Directives

Each program, service program, and module is associated with a set of migration parameters called **Directives** that control the migrated output for the individual program. The image below shows the migration directives for a program. The directives for a service program and module are similar except for the **Remote** section noted in the narrative below.

Internal

When the application calls this program anywhere in the code, this option directs the Agent to produce an internal call.

Migrate checkbox indicates the program becomes part of the list of programs to migrate when the GamePlan is migrated. The *Class name* is the name of the class that implements in the new code when the program is internal to the application. This is the opportunity to modernize the naming of the programs in the migrated code. When the name is changed, any program that calls this program will use the new name.

When *Make /COPY paths relative to the location of this program's source file location* is checked, the migration agent uses the common root of the program's source path and the copybook path to issue a relative path reference for the /COPY statement.

Remote

> ***Note:*** *The directives for a service program or module do* ***NOT*** *include this option.*

When the application calls this program anywhere in the code, this option directs the Agent to produce a remote call to a program running on the System i.

The name of the *Library* where the program exists when the program stays on the iSeries, *LIBL can be used as the library list or iSeries library name. *Program name* is the name of the iSeries PGM object used in all remote calls to this program.

External

When a program is external to the application, the .Net runtime needs to load a Dynamically Linked Library Assembly (DLL), locate the class in the namespace, instantiate it, and then call one of its methods.

Namespace is where the external program resides, *Class name* represents the called program, usually the "Class Library" (DLL) in which the external program will reside. *Assembly name* is the namespace Assembly, which defaults to *Namespace.ClassName*.

Use the *Replace program call with a method name* checkbox if the program call is to be replaced with the *Method name* everywhere this program is referenced.

Op-Code Support

All RPG Op-codes are supported except for the following: ACQ, ALLOC, DEALLOC, DEBUG, DSPLY, DUMP, NEXT, POST, REALLOC, REL, RESET, and SHTDN.

Normalization of RPG Op-codes

Some RPG op-codes have gone through small syntax changes as revisions of the language evolve. The following table shows the list of op-codes normalized before migration starts.

Deprecated	Normalized	Deprecated	Normalized
BITOF	BITOFF	PARM	DCLPARM
CAT	CONCAT	OCUR	OCCUR
CHEKR	CHECKR	REDPE	READPE
COMIT	COMMIT	RETRN	RETURN
COMP	COMPARE	ROLBK	ROLLBACK
DEFN	DEFINE	SELEC	SELECT
EXTRCT	EXTRACT	SORTA	SORTARR
EXCPT	EXCEPT	SUBST	SUBSTR
DELET	DELETE	TEST	TESTTIME
DOU	DOUNTIL	TESTB	TESTBITS
DOW	DOWHILE	TESTN	TESTNUM
ITER	ITERATE	UNLCK	UNLOCK
LOKUP	LOOKUP	UPDAT	UPDATE
MOVEA	MOVEARR	Z-ADD	ZADD
MVR	MOVEREM	Z-SUB	ZSUB

Loading Program References

The first thing the RPG Agent does is to load all the program references, which are:

1. List of Programs called.
2. List of Files used.
3. List of Modules used.
4. List of Menus used (partially supported).
5. List of Data Areas used.
6. List of Copybooks referenced.
7. List of Third Party System i object calls.
8. List of Message Files referenced.

Monarch needs the Directive page of each program it calls to have all the information about how to generate the new code to execute the program. Most of the programs end up as internal calls to a *.Net class.*

Once the references are loaded, the RPG Agent will attempt to locate all the objects referred to by this program in the GamePlan. If any of the references are missing, a **Task** will be issued but the migration will proceed.

Generation of Migrated Code - Behind the Scenes

/Error AVR compiler directive

Monarch will generate as much code as possible, even if there are tasks to be resolved. Most of the resolution of these tasks is done by changing or adding new AVR code in Visual Studio to implement unsupported features.

Every time the RPG Agent needs to capture the attention of the Migrator, it will insert a line that starts with /Error. The AVR compiler will stop at these /Error directives and display the message next to it; the same way it would handle any other compiler error.

The Migrator should read the message Monarch generated, analyze it, design the code that needs to be added or changed, *remove* the /Error directive line, and then re-compile the application.

Migrated Visual RPG File's Header

```
// Migrated on 2/14/2008 at 3:36 PM by ASNA Monarch(R) version 4.0.16
// Migrated source location: library JB_M6CUST, file QRPGSRC, member CUSTDELIV
Using System
Using ASNA.Monarch
```

The RPG Agent will start writing a header with a comment that contains Monarch version information and the timestamp as shown above.

Immediately following, a list of Using directives are included that are required to avoid fully qualified method calls.

If the Migrator wishes to modify this list, the existing RPG template file may be modified or a new Template file created.

> ***Note:*** *See Establishing Templates and Cascading Style Sheets in Chapter 5 for special considerations concerning the modification of Monarch Cocoon Templates.*

If the RPG template is modified or a new RPG template is created, Cocoon needs to be re-started in order for the RPG Agent to use the new files.

New RPG Program's Wrapper Class

```
BegClass ProgramName Extends(ASNA.Monarch.Program) Access( *Public )
    BegProc *Entry Access( *Public )
       // This is where the legacy migrated code will be produced.
       // The new code in this section is almost identical
       // to the original code, except for the fact that it uses free format syntax.
     EndProc
    BegConstructor Access( *Public )
       . . .
    EndConstructor
    BegSr Dispose Access(*Public) Modifier(*Overrides)
       . . .
        *Base.Dispose(Disposing)
    EndSr
EndClass
```

* Every Migrated Program becomes a Class

All programs in the Monarch framework are classes derived from ASNA.Monarch.Program.

* The new Program's constructor opens DB and Initializes Fields

When a program running under the Monarch framework calls another program and there is no active program in memory, 1) an instance of this program gets created (see **Entry procedure* below), and 2) an initialization subroutine named *constructor* runs. Monarch generates code to perform the following functions:

1. *Implicitly* read Data-Area data structures
2. Loads compile-time data arrays
3. Opens database files

* The new Program's "*destructor*" closes DB

When the Monarch framework needs to terminate a program, the subroutine Dispose is called. For the implementation of the Dispose subroutine, Monarch will add code to perform the inverse of the code in the constructor, namely,

1. *Implicitly* write Data-Area data structures
2. Close database files

* The BegProc **Entry* Procedure

The BegProc *Entry procedure makes the conversion between procedural RPG and an Object-Oriented Visual RPG preserving the legacy code as close to the original as possible.

The BegProc *Entry procedure implements the Activation and Invocation of programs just like on the iSeries. If a program leaves *INLR set when it completes executing, the program is *destroyed* (removed from memory), so that the next time other programs call this program, a brand new instance will be created. Otherwise, the program remains in memory for future *activations*.

ASNA.Monarch.Program Base Class

All RPG programs are migrated as a class that extends the ASNA.Monarch.Program base class.

By virtue of being derived from ASNA.Monarch.Program, the migrated program inherits support for the following areas:

1. Data Area support (including LDA)
2. Program message support
3. Remote command execution support (access to programs that stay in the System i)
4. Access to **monarchJob** (which in turn makes any public subroutine and function defined in the job accessible to the program)

Migrated C-Specs, which are not directly supported by Visual RPG, will be translated into calls to the subroutines defined in ASNA.Monarch.Program, and its fields and/or calls to the associated Job class.

The subroutines and fields defined in ASNA.Monarch.Program are accessible to the Migrator when resolving unsupported features or modernizing the application. It is also possible to develop additional support into a class derived from ASNA.Monarch.Program and later change all programs to be derived from it; effectively increasing functionality to all of your migrated programs.

```
BegClass MyShopBaseProgram Extends(ASNA.Monarch.Program) Access(*Public)
```

```
BegClass LegacyProgram_1 Extends(MyShopBaseProgram) Access(*Public)
```

```
BegClass LegacyProgram_2 Extends(MyShopBaseProgram) Access(*Public)
```

ASNA.Monarch.Job class

You will notice in the new migrated code that references to fields and or subroutines are done through ASNA.Monarch.Job. The Monarch Job class implements the support for the JOB on the iSeries. ASNA.Monarch.Program defines a field called *monarchJob*, which is an instance to ASNA.Monarch.Job.

Monarch-Generated Properties

Monarch makes extensive use of properties (**BegProp/EndProp** functions).

For migration purposes, properties accomplish the following desirable goals:

Allow implementation of read-only fields

For example, it is desirable that *INLR is made public, but only for reading.

With properties, it is possible to implement a read-only field. For example, the next piece of code implements a property where only the Get access is provided. Leaving out the **BegSet/EndSet** effectively makes StarIndicatorLR a read-only field.

```
BegProp StarIndicatorLR Type(*Ind)
    BegGet
        LeaveSr *INLR
    EndGet
EndProp
```

Implement a solution to the overlapping data-structure field challenge

Take for example the following overlapping data-structure. It defines three overlapping fields: #JSCDS a Char(22), and two Zoned(11,2) fields that overlap #JSCDS.

```
                DS
#JSCDS              1    22
W9DEBT              1    11 2
W9CRED             12    22 2
```

Monarch uses the following constructs, making use of properties and DclAlias AVR op-code.

```
DclDS
  DclDsFld #JSCDS  Type( *Char ) Len( 22 )
  DclAlias W9DEBT Exp(Prop_W9DEBT) Type( *Zoned    ) Len( 11,2 )
      BegProp Prop_W9DEBT Type( *Zoned    ) Len( 11,2 )
        BegGet
          LeaveSR #JSCDS.Substring(0,11)
        EndGet
        BegSet
          Move     *PropVal   _W9DEBT Type(*Char) Len(11)
          #JSCDS = _W9DEBT + #JSCDS.Substring(11)
        EndSet
      EndProp
  DclAlias W9CRED Exp(Prop_W9CRED) Type( *Zoned    ) Len( 11,2 )
      BegProp Prop_W9CRED Type( *Zoned    ) Len( 11,2 )
        BegGet
          LeaveSR #JSCDS.Substring(11,11)
        EndGet
        BegSet
          Move    *PropVal _W9CRED Type(*Char) Len(11)
          #JSCDS = #JSCDS.Substring(0,11) + _W9CRED
        EndSet
      EndProp
```

Fortunately, thanks to Visual Studio collapsible-regions, when the code is collapsed, it looks more readable, as shown:

This is really close to the original code, but "under the covers", the implementation (using properties) can sub-string chars from #JSCDS to store W9DEBT and W9CRED.

Improve code readability

In several places, Monarch will make use of properties to simplify fully qualified names with a shorter more readable name as shown in the next example.

```
BegProp _Monarch_USER Type(*Char)  Len(10)
    BegGet
        LeaveSr MonarchJob.PsdsJobUser
    EndGet
    BegSet
        MonarchJob.PsdsJobUser=*PropVal
    EndSet
EndProp
```

The name _Monarch_USER may be used instead of MonarchJob.PsdsJobUser.

Local Data Area

Each ASNA.Monarch.Job instance defines a block of memory for the LDA. The LDA is a System.Text.StringBuilder field type, initialized to 1024 blank *Char.

The Job provides to methods to store and retrieve data from the LDA:

```
System.String SetLdaField(int start, int length, string newValue)
GetLdaField(int start, int length)
```

To simplify, Monarch.Program provides the same methods as the Job, by calling the Job's methods through monarchJob:

```
BegFunc GetLdaField Type(*String)
   DclSrParm start  Type(*Int32)
   DclSrParm length Type(*Int32)
   LeaveSr monarchJob.GetLdaField( start, length )
EndFunc
BegSr SetLdaField Access(*Protected)
    DclSrParm start Type(*Int32)
    DclSrParm length Type(*Int32)
    DclSrParm newValue Type(*String)
    monarchJob.SetLdaField( start, length, newValue )
EndSr
```

The following is a section of a CL program that deals with LDA, along with its corresponding Monarch-generated code snippet:

```
/*--------------------------------------------------------------*/
/* MOVE SYSTEMS DATA AREA TO LDA                          */
/*--------------------------------------------------------------*/
        CHGDTAARA  DTAARA(*LDA (1 50)) VALUE(&SYSTEMS)
        CHGDTAARA  DTAARA(*LDA (51 10)) VALUE(&JOBNAME)
        CHGDTAARA  DTAARA(*LDA (61 10)) VALUE(&USER)
        CHGDTAARA  DTAARA(*LDA (101 2)) VALUE(&CENTURY)
        CHGDTAARA  DTAARA(*LDA (71 6)) VALUE(&JOBNBRA)
```

```
*--------------------------------------------------------------*/
/* MOVE SYSTEMS DATA AREA TO LDA                         */
/*--------------------------------------------------------------*/
   SetLdaField(1, 50, _SYSTEMS)
   SetLdaField(51, 10, _JOBNAME)
   SetLdaField(61, 10, _USER)
   SetLdaField(101, 2, _CENTURY)
   SetLdaField(71, 6, _JOBNBRA)
```

System i Data Area access

Except for the LDA, Monarch does not provide an equivalent .Net construct. Monarch will generate code to remotely access the Data Areas back on the System i.

If the Migrator needs to move the Data Areas from the System i, the Monarch-generated code (described below) needs to be replaced with new subroutines that implement a .Net serialization for these named blocks of memory. Possibly the most simple implementation would be to use the System.Xml framework support and implement these Data Areas as XML files stored on the server.

ASNA.Monarch.Program provides the following Data Area support. There is also additional support provided by ASNA.Monarch.CLProgram:

```
System.String DataArea_In(System.String library, +
 System.String dataarea, *Boolean withLock)
DataArea_Out(System.String library, System.String dataarea, +
 System.String newValue, *Boolean keepLock)
DataArea_Unlock(System.String library, System.String dataarea )
```

Keeping *LDA on the iSeries

When a migrated application uses a remote call to programs back on the iSeries, and those programs rely on the *LDA, the Migrator must be careful to replace calls:

From:

```
GetLdaField
SetLdaField
```

To:

```
DataArea_In( '', '*LDA', *False)
DataArea_Out('', '*LDA', newValue, *False)
```

Additional Monarch-Generated Data Area Related Subroutines

When the DEFINE operation code is used to define a Data Area, (Factor 1 set to *NAMVAR or *DTAARA), Monarch will generate two additional subroutines:

```
RetrieveAllDataAreas (*Boolean withLock)
ChangeAllDataAreas   (*Boolean withLock)
```

These will be called to read all the data areas or write to all data areas where appropriate. As shown by the next code snippet, the implementation is added at the end of the file in two regions, where the body of each subroutine is defined by multiple calls to DataArea_In and DataArea_Out:

```
/region Retrieve All Data Areas
   BegSr RetrieveAllDataAreas
      DclSrParm WithLock *Boolean
      CNTRY1 = DataArea_In('*LIBL','CNTRY1',WithLock)
      CNTRY2 = DataArea_In('*LIBL','CNTRY2',WithLock)
      CENTYR = DataArea_In('*LIBL','CENTYR',WithLock)
  EndSr
/endregion
/region ChangeAll Data Areas
  BegSr ChangeAllDataAreas
      DclSrParm WithLock *Boolean
      DataArea_Out('*LIBL','CNTRY1',CNTRY1, WithLock)
      DataArea_Out('*LIBL','CNTRY2',CNTRY2, WithLock)
      DataArea_Out('*LIBL','CENTYR',CENTYR, WithLock)
  EndSr
/endregion
```

As discussed before, additional calls to these Monarch-generated routines may be added at any place in the program when adding or resolving unsupported features.

Free Format C-Specs

When the RPG Migration Agent encounters free format C-specs, every attempt is made to translate the free format into valid AVR code. The free format C-specs will be subjected to the same migration validation as formatted RPG.

The next examples shows free format C-specs followed by the code generated by the RPG Migration Agent.

```
0494.00     C   DeleteCust    BEGSR
0496.00     /Free
0497.00          CMCUSTNO   = SFCUSTNO;
0498.00          LockCust   = 'Y';
0499.00          Exsr CustChk;
0500.00          CusWasDltd = *Off;
0501.00          DLTCSTZIP  = %Trim(CMNAME);
0503.00          ExFmt CONFDLT;
0505.00          If CancelKey = *On;
0506.00             Unlock  CUSTOMERL1;
0507.00          EndIf;
0508.00      /end-free      )
0545.00     C              ENDSR
```

```
BegSr DeleteCust Access( *Private )
    CMCUSTNO    = SFCUSTNO
    LockCust    = 'Y'
    Exsr CustChk
    CusWasDltd = *Off
    DLTCSTZIP  = %Trim(CMNAME)
    ExFmt CONFDLT
    If CancelKey = *On
      Unlock        CUSTOMERL1
    EndIf
. . .
EndSr
```

Status Data Structures

The RPG Agent generates code in every migrated program to support:

1. Program Status Data Structure
2. File Information Data Structure

Program Status Data Structures

The following shows the properties generated for each of the supported fields.

Parameter or Keyword	Populated Program Status Data Structure
Program	*this.GetType().FullName.Substring(*this.GetType(). + Namespace.Length(1).PadRight(10).ToUpper()
Parameters	See "***Number of Parameters Subfield in PSDS***" below
Job Start Date	MonarchJob.StartupMoment
Job	MonarchJob.PsdsJobName.ToUpper()
User Id	MonarchJob.PsdsJobUser.ToUpper()
Job Number	MonarchJob.PsdsJobNumber
Job Start Time	MonarchJob.StartupMoment
Program Start Date	StartupMoment
Program Start Time	StartupMoment
*PROC	Same as Program
*PROGRAM	Same as Program

The following is a sample of a typical PSDS as defined in Legacy Source.

```
D@@pgm_ds        sds
D  @@prms                 37     39  0
D  @@job                 244    253
D  @@user                254    263
D  @@jobn                264    269
D  @@date                276    281  0
D  @@time                282    287  0
```

ASNA Monarch would generate a Data Structure as follows.

```
/region Program Status Data Structure
    DclDs @@pgm_ds
        DclDsFld QRPGCLESRC_MonarchName51 Type ( *Char) Len( 36 )
        DclDsFld @@prms              Type( *Zoned    )     Len( 3,0 )
        DclDsFld QRPGCLESRC_MonarchName52 Type ( *Char ) Len( 204)
        DclDsFld @@job               Type( *Char     )     Len( 10  )
        DclDsFld @@user              Type( *Char     )     Len( 10  )
        DclDsFld @@jobn              Type( *Char     )     Len( 6   )
        DclDsFld QRPGCLESRC_MonarchName53 Type ( *Char ) Len( 6)
        DclDsFld @@date              Type( *Char    )     Len( 6,0 )
        DclDsFld @@time              Type( *Char    )     Len( 6,0 )
/endregion
```

In the BegConstructor region, the following code is generated:

```
/region Populate Program Status Data Structure
  @@job = MonarchJob.PsdsJobName.ToUpper()
  @@user = MonarchJob.PsdsJobUser.ToUpper()
  @@jobn = MonarchJob.PsdsJobNumber
  @@date = StartupMoment
  @@time = StartupMoment
/endregion
```

Number of Parameters Subfield in PSDS

There is special consideration given to the number of parameters subfield (@@prms in the above data structure). When this subfield is encountered, ASNA Monarch sets the *Parms built-in function to the value of the subfield in the BegProc section of the *Entry program. For example:

```
BegProc *Entry Access(*Public)
   DclSrParm qp_key    By(*Reference) Options(  *NoPass )
   DclSrParm qp_show   By(*Reference) Options(  *NoPass )
   @@prms = *Parms
   tbhkey = qp_key
   If( %parms >= 2 *And %addr(qp_show) <> *null )
       q_show = qp_show
   Else
       q_show = '*NO'
   EndIf
EndProc
```

File Information Data Structure

Monarch's implementation of the File information is not a true data structure per se, but a list of read-only properties that contain the values of the respective status fields.

The properties generated are those that the particular migrated program refers to in the data structure for each specified file. The properties are distinct by using the file name given in the F-Specs. The following table lists the supported File Information Status Data fields:

Offset or Keyword	Description (Property)
156	**Record Count (RecCount)
261	Record
367	Flags (FeedbackFlags)
369	AID Byte (FeedbackAID)
370	Cursor Row (FeedbackCursor / H'100')
371	Cursor Column (FeedbackCursor*BitAnd H'FF')
376	Relative Record Number (RecNum)
*FILE	File
*RECORD	Record Name

*** Record count supported for disk files only.*

The following example legacy source shows three information file data structures. One for a workstation file (SDP101D) and one for a database file (PS100L01).

```
     FSDP101d   cf   e              Workstn infds(@F$$)
     FPS100L01  uf a e            k disk    usropn
     F                                      infds(@FF1)
     D @F$$             DS
     D  f@FnKy                369     369
     D  f@LIN#                370     370
     D  f@COL#                371     371
     D  f@CSRL                370     371B 0
     D  f@SFL#                376     377B 0
     D @FF1             DS
     D  F1@RAO                156     159B 0
     D  F1@RR#                397     400B 0
```

Monarch would generate the necessary code to set these properties in the "File Information Data Structures" region as shown below. Notice how the name of the properties for the workstation file and the database file refer to SDP101D and PS100L01 respectively.

```
/region File Information Data Structures

    BegProp f@FnKy Type(*Byte)
        BegGet
            LeaveSr SDP101D.FeedbackAID
        EndGet
        BegGet
            SDP101D.FeedbackAID = *PropVal
        EndSet
    EndProp
    BegProp f@LIN# Type(*Byte)
        BegGet
            LeaveSr SDP101D.FeedbackCursor/ H'100'
```

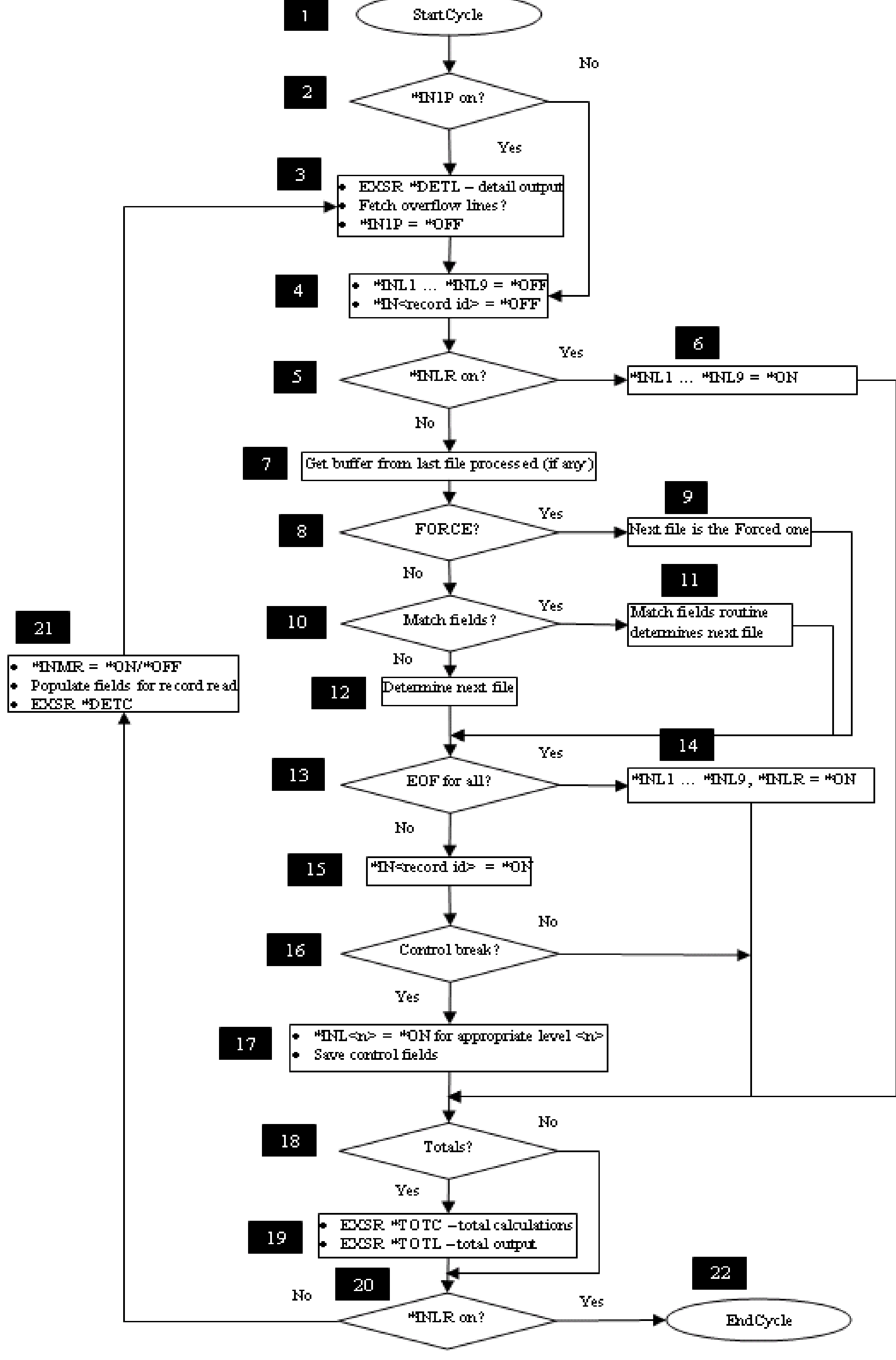
```
        EndGet
    EndProp
    BegProp f@COL# Type(*Byte)
        BegGet
            LeaveSr SDP101D.FeedbackCursor*BitAnd H'FF'
        EndGet
    EndProp
    BegProp f@CSRL Type(*String)
        BegGet
            LeaveSr SDP101D.FeedbackCursor
        EndGet
    EndProp
    BegProp f@SFL# Type(*String)
        BegGet
            LeaveSr SDP101D.SflRrn
        EndGet
    EndProp
    BegProp F1@RAO Type(*Integer) Len(4)
        BegGet
            LeaveSr PS100L01.RecCount
        EndGet
    EndProp
    BegProp F1@RR# Type(*Integer) Len(4)
        BegGet
            LeaveSr PS100L01.RecNum
        EndGet
    EndProp
/endregion
```

Cycle Support

Overview

The RPG Cycle is supported in RPG ILE applications in the AVR Cycle. The AVR Cycle is a series of ordered steps that the main procedure goes through for each record read. Depending on the coded specifications, your program may or may not use each step in the cycle. This topic only deals with the AVR Cycle and does not attempt to compare the AVR and RPG Cycles.

AVR Cycle



Cycle Narrative

1. *StartCycle is the name of the subroutine that performs the Cycle. It assumes the primary and all secondary files are already open, which causes the first record from the primary and all secondary files to be read, and all program fields have their correct values including first page indicator (*IN1P is initialized to *ON in the class constructor).
2. If *IN1P is *ON (this is the first time *StartCycle is called), go to step 3. Otherwise, go to step 4.
3. Perform header and detail output (HeadingSpec and DetailSpec). Set *OFF *IN1P.
4. Set *OFF all record identifying and L1 through L9 indicators.
5. If *INLR is *ON, go to step 6, otherwise go to step 7.
6. Set *ON the level indicators *INL1 through *INL9, and go to step 18.
7. If this is not the first program cycle, read a record from the last file processed.
8. If FORCE was issued on the previous cycle, go to step 9, otherwise go to step 10.
9. The forced file is selected for processing. If the forced file is at EOF, go to step 10. Otherwise, the match record indicator (*INMR) is set to *OFF and the match fields in the forced file are saved. Continue at step 13.
10. If match fields are used in the program, go to step 11. Otherwise, go to step 12.
11. The match field routine selects the next file. Continue at step 13.
12. The next file is selected. It will be either the primary if it is not at EOF, or the first secondary that is not at EOF selected in the order by which they are specified in the program.
13. If all files are at EOF go to 14, otherwise go to 15.
14. Set *ON *INLR and all level indicators *INL1 through *INL9. Continue at 18.
15. The record identifying indicator is set *ON for the record selected for processing.
16. If there is a control break go to 17, otherwise go to 18.
17. The appropriate control level indicator (*INL1 through *INL9) is set *ON. All lower level indicators are also set *ON.
18. Determine whether totals need to be executed, and if so go to 19 otherwise go to 20. If no control levels are specified for any record, totals are bypassed on the first cycle and are always processed after the first cycle. If control levels are specified, totals are bypassed until the first record containing control fields has been processed.
19. Total Calculations (*TotalCalc) and total output (TotalSpec) are processed.
20. If *INLR is *ON, go to step 22.
21. The match record indicator (*INMR) is set *ON or *OFF depending on whether the record read is a matching record. Data from the last record read is made available for processing. Detail Calculations (*DetailCalc) are processed. Continue at step 3.
22. Return to the caller.

File Designation Keyword

RPG Agent generates "Designation" keyword.

For example:

CHERRY PP_MNRCH/QRPGSRC/MUTSVC_3 contains the following four files.

```
0127.00      FPCBLOT3PIP  E           K        DISK
0128.00SK01 FBLOTPRTPIF  E           K        DISK
0129.00AR02 FPCBANNECIS  E           K        DISK
0130.00MC04 FPRSHLPS  IS  E           K        DISK
```

The following shows these files when migration is completed.

```
DclDiskFile PCBLOT3P  Type(*Input) Org(*Indexed) DB(MyJob.MyDatabase) +
    File("*LIBL/PCBLOT3P") ImpOpen(*No) Designation(*Primary)
DclDiskFile BLOTPRTP  Type(*Input) Org(*Indexed) DB(MyJob.MyDatabase) +
    File("*LIBL/BLOTPRTP") ImpOpen(*No)
DclDiskFile PCBANNEC  Type(*Input) Org(*Indexed) DB(MyJob.MyDatabase) +
    File("*LIBL/PCBANNEC") ImpOpen(*No) Designation(*Secondary)
DclDiskFile PRSHLPS   Type(*Input) Org(*Indexed) DB(MyJob.MyDatabase) +
    File("*LIBL/PRSHLPS")  ImpOpen(*No) Designation(*Secondary)
```

*StartCycle

This method is used to signify the start of the AVR cycle, without which, the program is not considered a cycle program.

```
BegProc *Entry Access(*Public)
   ExSr *StartCycle
EndProc
```

First Page

Before the first record is read the first time through the cycle, the program resolves any parameters passed to it, writes the records conditioned by the 1P (first page) indicator, and processes any heading or detail output operations having no conditioning indicators.

For example, heading lines printed before the first record is read might consist of constant or page heading information or fields for reserved words, such as PAGE and UDATE.

The first page indicator is turned on in the class constructor and turned off after the first page headings (if any) are printed. In the following example, one HeadingSpec QPRINT_DETH1P would be printed as it is conditioned by *In1P (first page indicator).

```
BegCyclePrintAttr
    HeadingSpec QPRINT_DETH1P    Cond(*InOF *Or *In1P )
    HeadingSpec QPRINT_DETH1L1F  Cond(*InL1 ) FetchOverflow(*Yes)
    DetailSpec  QPRINT_DET01     Cond(*In01 )
    TotalSpec   QPRINT_TOTAL2LRF Cond(*InLR ) FetchOverflow(*Yes)
EndCyclePrintAttr
```

Matching Records

To process matching records in AVR, the fields defined as the matching record fields are assigned using **M1…M9** on the DclFmtCycleAttr command for each file format on which the matching is to occur. For instance, if you have an order master file, an order detail file, and a backorder file matched by their respective order numbers, you would specify the order number as the matching field in each files as shown here.

```
DclFmtCycleAttr RMaster   *in77 M1(RMOrder#)
DclFmtCycleAttr RDetail   *in42 M1(RDOrder#)
DclFmtCycleAttr RBoOrders *in43 M1(RBOrder#)
```

The AVR Cycle provides a hold area for each file that has match fields. The match fields are extracted from the files and the values tested to determine which file is to be processed next. When a match is found, indicator ***INMR** is set on.

The next record is selected for processing based on the value in the match fields AND either a primary or the next secondary according to the order declared.

> ***NOTE: If the match fields are not in sequence - NO exception or error handling is performed in the AVR Cycle - You must add processing to the migrated code for sequence checking.***

In the example below, there are two secondary files - *Detail* and *BackOrd*. The order in which they are declared makes *Detail* the first secondary file and *BackOrd* the second secondary file.

```
DclDiskFile Master  Designation(*primary)   Type(*input)
DclDiskFile Detail  Designation(*secondary) Type(*input)
DclDiskFile BackOrd Designation(*secondary) Type(*input)
```

Level Breaks

To define level breaks, the control fields are assigned using **L1…L9** on the DclFmtCycleAttr command for each file format on which the level breaks are to occur. For instance, you have an order master file (one record per order), an order detail file (one record per item in the order), and a backorder file (multiple records per order and item). Specify the order number as the level break field in each file and include the item level on the backorder file.

Notice how RBoOrders file is set with L1 and RMaster is set with L2. When L1 is triggered, L2 is set on also. This would apply to all lower level indicators that may be present as well.

```
DclFmtCycleAttr RMaster   *in77 M1(RMOrder#) L2(RMOrder#)
DclFmtCycleAttr RDetail   *in42 M1(RDOrder#)
DclFmtCycleAttr RBoOrders *in43 M1(RBOrder#) L1(RBOrder#, RBItem#)
```

*TotalCalc

***TotalCalc** (method) indicates the beginning of the Total calculations subroutine. Code within the subroutine will contain those RPG C specs where the RPG Control indicator columns were used. For example this code,

```
0053.00      **** Customer number Change - Total Time
0054.00      CL0                    If        *INL1 = *On
0055.00      CL0                    Eval      #TotalAmt = #TotalAmt + #L1TotAmt
0056.00      CL0                    EndIf
```

Becomes this (control indicator LO used in the C spec):

```
  BegSr *TotalCalc
  //      **** Customer number Change - Total Time
     If( *INL1 = *On )
        #TotalAmt = #TotalAmt + #L1TotAmt
     Endif
  EndSr
```

*DetailCalc

***DetailCalc** (method) indicates the beginning of the Detail calculations subroutine. Code within the subroutine will contain those RPG C specs where the RPG Control indicator columns were blank. For example this code,

```
  **** Customer Number Change
  C                   If        *INL1 = *On
  C                   Clear                   #L1TotAmt
  C     OrdCustNo     Chain     CMastNewL1                         90
  C                   EndIf
   **** Process Detail Record
  C                   If        *IN01 = *On
  C                   Eval      #L1TotAmt = #L1TotAmt + OrdAmt
  C                   Eval      #OrdDteUSA = %date(OrdDTE:*USA0)
  C                   Eval      ShrtName = %TrimR(CMName)
  C                   Eval      ShrtAddr = %TrimR(CMAddr1)
  C                   Move      CMPostCode    #ZipChar
  C                   Eval      #CtStZip = %TrimR(CMCity) + ', ' + CMState +
  C                             '  ' + #ZipChar
  C                   EndIf
```

Becomes this (the control indicator in the C spec is blank):

```
BegSr *DetailCalc
If ( *INL1 = *On )
    Clear #L1TotAmt
    Chain CMastNewL1 Key(OrdCustNo) Err(*In90)
EndIf
  //      **** Process Detail Record
If( *IN01 = *On )
    #L1TotAmt = #L1TotAmt + OrdAmt
    #OrdDteUSA = %date(OrdDTE:*USA0)
    ShrtName = %TrimR(CMName)
    ShrtAddr = %TrimR(CMAddr1)
    Move CMPostCode #ZipChar
    #CtStZip = %TrimR(CMCity) + ', ' + CMState +'  ' + #ZipChar
Endif
EndSr
```

Last Record

During the last time a program goes through the cycle, when no more records are available, the LR (last record) indicator and all level indicators (L1 through L9) are set on. Any total calculation and total output are completed and then control is returned to the caller.

In the following example, the two print lines shown are conditioned by *InL1 and *InLR which will be printed when the last cycle is completed.

```
    HeadingSpec QPRINT_DETH1L1   Cond(*InL1 )
    TotalSpec   QPRINT_TOTAL2LRF Cond(*InLR ) FetchOverflow(*Yes)
```

Cycle Printing

BegCyclePrintAttr / EndCyclePrintAttr section is automatically created to control the cycle printing. Within this print cycle code there will be **TotalSpec**, **DetailSpec**, and **HeadingSpec** definitions representing the O-spec total, detail, and heading lines respectively. Notice how each of these include the conditions under which each will be printed.

```
BegCyclePrintAttr
    HeadingSpec QPRINT_DETH1P    Cond(*InOF *Or *In1P )
    HeadingSpec QPRINT_DETH1L1   Cond(*InL1 )
    HeadingSpec QPRINT_DETH2L1F  Cond(*InL1 ) FetchOverflow(*Yes)
    DetailSpec  QPRINT_DET01     Cond(*In01 )
    TotalSpec   QPRINT_TOTALL1   Cond(*InL1 )
    TotalSpec   QPRINT_TOTAL2LRF Cond(*InLR ) FetchOverflow(*Yes)
EndCyclePrintAttr
```

See Monarch-Generated Record and Field Names in Chapter 10 for more information on the migration and printing of print files not contained within an AVR Cycle.

Overflow

The fetch overflow routine allows you to alter the overflow logic to prevent printing over the perforation and to let you use as much of the page as possible. Fetch overflow in legacy is specified with an F in position 16 of the output specifications on any detail, total, or exception lines for a PRINTER file.

Fetching for Overflow means that *before* printing the record format, the Overflow condition needs to be checked and if overflow printing would occur, the overflow indicator is set on. When the indicator is set, **all** lines **conditioned on overflow** will be printed first. In the example above, notice how QPRINT_DETH2L1F and QPRINT_TOTAL2LRF both have the FetchOverflow keyword while QPRINT_DETH1P is conditioned by the overflow indicator (*InOF) and if the overflow indicator is set *On, would be output prior to either of the fetching print lines.

Lookahead Fields

Lookahead fields are not supported but code can be added to the migrated *DetailCalc section to add the look ahead functions.

Examples

Legacy Source

```
****************************** Beginning Of Data *********************
    ****            Print Example.           ****
    ***************************************************
    FORDFile   IP   E             Disk
    FCMastnewL1IF   E           K Disk
    FQPRINT    O    F  132        Printer OFLIND(*INOF)

    D                SDS
    D #PgmName          *Proc

    D@Header1          C                   'Order Detail Rpt'
    D@L1Head1          C                   'Customer #'
    D@L1Head2          C                   'Name/Address'
    D@L1Head3          C                   'Phone'
    D@L1Head4          C                   'Order Detail'
    D@L1Head5          C                   'Order Number'
    D@L1Head6          C                   'Order Date'
    D@L1Head7          C                   'Order Qty'
    D@L1Head8          C                   'Order Amt'
    D@L1Tot1           C                   '----------------'
    D@LRTot1           C                   '================'
    D@LRTot2           C                   'Grand Total:'

    D#L1TotAmt         S              11  2
    D#TotalAmt         S              11  2
    D#TimeDate         S              14  0
    D#OrdDteUSA        S                D   DatFmt(*USA)
    D#CtStZip          S              31
    D#ZipChar          S               5
    DShrtName          S              20
    DShrtAddr          S              20

    IRORDFile      01
    I                                        OrdCustNo     L1

    **** Customer Number Change
    C                   If        *INL1 = *On
    C                   Clear                   #L1TotAmt
    C     OrdCustNo     Chain     CMastNewL1                         90
    C                   EndIf

    **** Process Detail Record
    C                   If        *IN01 = *On
    C                   Eval      #L1TotAmt = #L1TotAmt + OrdAmt
    C                   Eval      #OrdDteUSA = %date(OrdDTE:*USA0)
    C                   Eval      ShrtName = %TrimR(CMName)
    C                   Eval      ShrtAddr = %TrimR(CMAddr1)
    C                   Move      CMPostCode    #ZipChar
    C                   Eval      #CtStZip = %TrimR(CMCity) + ', ' +
    C                             CMState + '  ' + #ZipChar
    C                   EndIf

    **** Customer number Change - Total Time
    CL0                 If        *INL1 = *On
```

```
    CL0                Eval      #TotalAmt = #TotalAmt + #L1TotAmt
    CL0                EndIf
    C     *INZSR       BegSr
    C                  Time                    #TimeDate
    C                  EndSr

    **** Printer O-Specs
    OQPrint    H    1P                    2 01
    O         Or    OF
    O                     #PgmName            10
    O                     @Header1            35
    O                     #TimeDate           62 '  :  :  &  /  /    '
    O                                         70 'Page:'
    O              N90    Page1          z    76
    O          H    L1                    2
    O                     @L1Head1            11
    O                     @L1Head2            26
    O                     @L1Head3            54
    O          HF   L1                    1
    O                     CMCustNo            11
    O                     ShrtName            34
    O                     CMPhone             57
    O          HF   L1                    1
    O                     ShrtAddr            34
    O          HF   L1                    1
    O                     #CtStZip            45
    O          HF   L1                 1  1
    O                     @L1Head4            23
    O          HF   L1                 1  2
    O                     @L1Head5            14
    O                     @L1Head6            26
    O                     @L1Head7            38
    O                     @L1Head8            50
    O          D    01                    1
    O                     InvNo               14
    O                     #OrdDteUSA          26
    O                     OrdQty         J    38
    O                     OrdAmt         J    50
    O          T    L1                    1
    O                     @L1Tot1             50
    O          T    L1                    1
    O                     #L1TotAmt      J    50
    O                                         52 '*'
    O          TF   LR                 1  1
    O                     @LRTot1             50
    O          T    LR
    O                     @LRTot2             32
    O                     #TotalAmt      J    50
    O                                         53 '*'
********************************* End Of Data ***************************
```

Migrated Code

```
Using MyCompany.MyApplication.Cycle416a_Job
Using System
Using ASNA.Monarch
BegClass Cycl_prt_4 Extends(ASNA.Monarch.Program) Access(*Public) +
                    Attributes(ActivationGroup('*DFTACTGRP'))
    DclDiskFile ORDFile  Type(*Input) Org(*Arrival) DB(MyJob.MyDatabase) +
           File("*LIBL/ORDFile") ImpOpen(*No) RnmFmt( RORDFILE ) +
           Designation(*Primary)
    DclDiskFile CMastnewL1  Type(*Input) Org(*Indexed) DB(MyJob.MyDatabase) +
           File("*LIBL/CMastnewL1") ImpOpen(*No) RnmFmt( RCMMASTL1 )
    DclPrintFile QPRINT DB(MyJob.MyPrinterDB) File("CYCLE416\CYCL_PRT_4") +
           ImpOpen(*No) OverflowInd(*INOF)
    DclConst @L1Head2          Value('Name/Address')
    DclConst @LRTot2           Value('Grand Total:')
```

```
    DclConst @L1Head1          Value('Customer #')
    DclConst @L1Head4          Value('Order Detail')
    DclConst @L1Head3          Value('Phone')
    DclConst @L1Head8          Value('Order Amt')
    DclConst @LRTot1           Value('================')
    DclConst @L1Head5          Value('Order Number')
    DclConst @L1Head6          Value('Order Date')
    DclConst @Header1          Value('Order Detail Rpt')
    DclConst @L1Head7          Value('Order Qty')
    DclConst @L1Tot1           Value('----------------')
/region Program Status Data Structure
    DclDs
        DclDsFld #PgmName          Type( *Char    )    Len( 10  )
/endregion
    DclFld #L1TotAmt         Type( *Packed  )    Len( 11,2 )
    DclFld #TotalAmt         Type( *Packed  )    Len( 11,2 )
    DclFld #TimeDate         Type( *Packed  )    Len( 14,0 )
    DclFld #OrdDteUSA        Type( *Date    )     TimFmt( *USA)
    DclFld #CtStZip          Type( *Char    )    Len( 31  )
    DclFld #ZipChar          Type( *Char    )    Len( 5   )
    DclFld ShrtName          Type( *Char    )    Len( 20  )
    DclFld ShrtAddr          Type( *Char    )    Len( 20  )
    DclFmtCycleAttr RORDFILE *IN01 L1(OrdCustNo)
    //     **** Customer Number Change
/region Constructor and Dispose
    . . .
/endregion
    BegProc *Entry Access(*Public)
        ExSr *StartCycle
    EndProc
    BegSr *DetailCalc
        If( *INL1 = *On )
            Clear #L1TotAmt
            Chain CMastNewL1 Key(OrdCustNo) Err(*In90)
        EndIf
    //     **** Process Detail Record
        If( *IN01 = *On )
            #L1TotAmt = #L1TotAmt + OrdAmt
            #OrdDteUSA = %date(OrdDTE:*USA0)
            ShrtName = %TrimR(CMName)
            ShrtAddr = %TrimR(CMAddr1)
            Move CMPostCode #ZipChar
            #CtStZip = %TrimR(CMCity) + ', ' + CMState +'  ' + #ZipChar
        EndIf
    EndSr
    BegSr PROCESS_STAR_INZSR Access( *Private )
        Time Target(#TimeDate)
    EndSr
    //     **** Printer O-Specs
    BegSr *TotalCalc
    //     **** Customer number Change - Total Time
        If( *INL1 = *On )
            #TotalAmt = #TotalAmt + #L1TotAmt
        EndIf
    EndSr
    BegCyclePrintAttr
        HeadingSpec QPRINT_DETH1P Cond(*In1P  *Or *InOF )
        HeadingSpec QPRINT_DETH1L1 Cond(*InL1 )
        HeadingSpec QPRINT_DETH2L1F Cond(*InL1 ) FetchOverflow(*Yes)
        DetailSpec QPRINT_DET01 Cond(*In01 )
        TotalSpec QPRINT_TOTALL1 Cond(*InL1 )
        TotalSpec QPRINT_TOTAL2LRF Cond(*InLR ) FetchOverflow(*Yes)
    EndCyclePrintAttr

EndClass
```

QPRINT Definition in Cocoon

Embedded SQL Migration

This topic contains three sections: Overview, Examples, and Monarch Framework Support.

The Overview provides detail statements of what happens to embedded SQL when it is migrated. The examples in this section show only snippets of code to demonstrate the statement.

The Examples section has complete legacy source and migrated code reflecting more fully the statements from the Overview section.

The Monarch Framework Support section shows the classes and detail of note to support the migration of the embedded SQL.

Overview

1. The RPG Agent assumes an ODBC driver for database connection when migrating the code. There are statements (in MyJob.vr) to define the connection field and the driver connection string (connectStr), assign the connection string value, and establish the connection. For example:

```
DclFld ADO_Connection Type(System.Data.Odbc.OdbcConnection) Access(*Public)
DclFld connectStr Type(*String)
 connectStr=string.Format("Driver={{Client Access ODBC Driver (32-bit)}};" + +
                             "System={0};DBQ={1},*USRLIBL;Uid={2};Pwd={3}", +
                             MyDatabase.Server, +
                             MyDatabase.DBName, +
                             MyDatabase.User,   +
                             MyDatabase.Password)
ADO_Connection = *New System.Data.Odbc.Odbc.Connection(connectStr)
```

   In MyJob.vr, the RPG Agent will place messages showing the replacement values when using MS SQL Server (instead of ODBC). When using MS SQL Server, the same statements will read as follows:

```
DclFld ADO_Connection Type(System.Data.SqlClient.SqlConnection) +
  Access(*Public)
DclFld connectStr Type(*string)
  connectStr = string.Format("Data Source={0};Initial Catalog={1};Integrated +
Security=True")
ADO_Connection = *New System.Data.SqlClient.SqlConnection(connectStr)
```

2. The SQL Commands directly supported are *PREPARE*, *DECLARE CURSOR*, *SELECT*, *OPEN CURSOR*, *CLOSE CURSOR*, *FETCH*, *INSERT*, *DELETE*, and *UPDATE*.

3. A "query" is migrated using one of the ExecuteSQL_Query methods (SqlQueryResults is the result set) while all others "non-query" SQL commands are migrated using the ExecuteSQL_Statement methods.

4. When *WHERE* clause host variables are used, the variable name in the legacy source is replaced within the migrated statement with a monarch-generated name as *"@sql_parm_n*" where '*n*' is a sequence number of the parameter in the original statement i.e. *@sql_parm_1*, *@sql_parm_2,* etc. The host variables' values are then passed to the methods as a collection of parameters using instances of DBParm, DBStrParm, or DBScaledParm classes.

For example, *WHERE wddate = :ODATE# and wdemp# = :F1EMP#* is migrated as shown below. Notice the two parameters @sql_parm_1 and @sql_parm_2 in the statement defined with DBParm and DBScaledParm respectively.

```
. . .( +
"WHERE wddate = @sql_parm_1 and wdemp# = @sql_parm_2;", + +
. . .
*New DBParm(DbType.Time, ODATE#),
*New DBScaledParm(DbType.Decimal, F1EMP#, %Len(F1EMP#), %DecPos(F1EMP#))
)
```

5. *INSERT* clause host variables are handled the same as host variables in the *WHERE* clause.

6. When the *SELECT* clause has host variables, the variables' values are converted to string and concatenated as part of the select statement.

For example, *SELECT :#TgtDate, :F4EMP#, WDORD#* is migrated as shown below. Notice how the two variables' values are converted to string using the built in function %Char when they are concatenated to the select statement.

```
SqlQueryResults = ExecuteSQL_Query ( +
"SELECT " + %Char(#TgtDate) + ", " + %Char(F4EMP#) + ", WDORD#, +
. . .
)
```

7. When the INTO keyword is used, the INTO section of the statement is removed and the values are propagated from the query results.

For example, *SELECT wddate, wdseq# INTO :F1DATE, :F1SEQ#* is migrated as shown below.

```
SqlQueryResults =  ExecuteSQL_Query ( +
"SELECT wddate, wdseq#                         " +
. . .
)
F1DATE = SqlQueryResults["wddate"] *AsFld F1DATE
F1SEQ# = SqlQueryResults["wdseq#"] *AsFld F1SEQ#
```

8. Select statements that have an expression (instead of a list) are modified by inserting "as SqlExprResultColumn" so that a new column is added to the result table, accessed as *SqlQueryResults["SqlExprResultColumn"]*.

For example, *SELECT IfNull(MAX(WDSEQ#),0)+1 INTO :F4SEQ#* is migrated as shown.

```
SqlQueryResults = ExecuteSQL_Query ( +
"SELECT IfNull(MAX(WDSEQ#),0)+1 as SqlExprResultColumn                  "  + +
. . .
)
F4SEQ# = SqlQueryResults["SqlExprResultColumn"] *AsFld F4SEQ#
```

9. The migration of a prepared statement is slightly more complex in that there are several commands to define the select statement, prepare the statement, declare the cursor, open the cursor (execute the statement), fetch from the result set, and ultimately close the cursor. However, once a statement is prepared it can then be executed multiple times within the scope of the source program. For simplicity, each of the following statements is taken individually as it relates to prepared statements.

For example, SelectF1 = 'select wddate, wdord#, wdseq# ' is migrated in a straightforward manner as shown.

```
SelectF1 = 'select wddate, wdord#, wdseq# '
```

For example, PREPARE SelF1 FROM :SelectF1 is migrated as shown. Notice how SelF1 is declared as Type(SqlPreparedStatement) then instanced with the connection and select statement (SelectF1).

```
DclFld SelF1 Type(SqlPreparedStatement)
SelF1 = *New SqlPreparedStatement( monarchJob.ADO_Connection, SelectF1 )
```

For example, DECLARE SelF1CSR SCROLL CURSOR FOR SELF1 is migrated as shown. Notice how SelF1CSR is declared as Type(SqlCursor) then instanced with the connection and prepared statement (SELF1). A scroll type is then assigned.

```
DclFld SelF1CSR Type(SqlCursor)
SelF1CSR = *New SqlCursor( monarchJob.ADO_Connection, SELF1 )
SelF1CSR.ScrollType = SqlCursor.ScrollTypes.Scrollable
```

For example, OPEN SelF1CSR is migrated as shown below. The open executes the SQL statement. The cursor (SelF1CSR) has its own SQLCA that is assigned to the global SQLCA so any testing of SQLCA.SQLCOD will still work. Although not shown, the RPG Agent will automatically qualify references to SQLCOD as SQLCA.SQLCOD since SQLCA is an instance of a class, and SQLCOD is a member field of SQLCA.

```
SelF1CSR.Open()
SQLCA = SelF1CSR.SQLCA
```

For example, FETCH NEXT FROM SelF1CSR INTO :F1DATE, :F1ORD#, :F1SEQ# is migrated as shown below. Notice how SelF1CSR.Fetch, which returns true when results are available, conditions the statements that retrieve the results preventing a program from accessing SqlQueryResults, (which is invalid (*nothing)), when the Fetch is unsuccessful.

```
If ( SelF1CSR.Fetch( SqlCursor.FetchOrientations.Next ) )
    F1DATE = SelF1CSR.SqlQueryResultsByIndex( 0 ) *AsFld F1DATE
    F1ORD# = SelF1CSR.SqlQueryResultsByIndex( 1 ) *AsFld F1ORD#
    F1SEQ# = SelF1CSR.SqlQueryResultsByIndex( 2 ) *AsFld F1SEQ#
EndIf
SQLCA = SelF1CSR.SQLCA
```

Notice how after the Fetch runs, the resulting data is retrieved using the name of the cursor (SelF1CSR) + "." + SqlQueryResultsByIndex, instead of the global SqlQueryResults (used for single-row queries). Each instance of SqlCursor class keeps its own SqlQueryResults collections, making it possible to use more than one cursor at a time while keeping the result datasets independent. The access by index as opposed to by name (as it is done when migrating single-row) is just a migration simplicity for the convenience of the RPG Agent, since the named collection used while preparing or creating the cursor may be out of scope.

Closing the cursor, which in the legacy source is something like, CLOSE SelF1CSR, now becomes:

```
SelF1CSR.Close
```

10. All other SQL commands not mentioned above will be blindly migrated as "ExecuteSQL_Statement (statement)". No attempt will be made to look for host-variables and no migration task will be produced (it is up to the data provider's implementation to accept or reject the request at application runtime).

Examples

The following examples give a better picture of how the legacy source is converted. At all times, always refer to the actual migrated code for the latest version.

Example - MyJob.vr

The following MyJob.vr shows the SQL connection established for embedded SQL. Notice the /Error statements generated as a reminder to change the code if MS SQL Server is used.

```
Using System
Using MyCompany.MyApplication
DclNamespace MyCompany.MyApplication.GroupAll_Job
BegClass MyJob Extends(ASNA.Monarch.WebJob) Access(*Public)
   DclDB Name(MyDatabase) DBName("*Public/DG NET Local") Access(*Public)
   DclDB Name(MyPrinterDB) DBName("DG Net Local") Access(*Public)
   DclFld ADO_Connection Type(System.Data.Odbc.OdbcConnection) Access(*Public)
/Error When using MS SQL Server, replace System.Data.Odbc.OdbcConnection with
System.Data.SqlClient.SqlConnection
    BegFunc getDatabase Type(ASNA.VisualRPG.Runtime.Database)+
            Access(*Protected) Modifier(*Overrides)
        LeaveSR MyDatabase
    EndFunc
    BegFunc getPrinterDB Type(ASNA.VisualRPG.Runtime.Database)+
            Access(*Protected) Modifier(*Overrides)
        LeaveSR MyPrinterDB
    EndFunc
    BegFunc getADO_Connection Type(System.Data.Common.DbConnection)+
            Access(*Protected) Modifier(*Overrides)
        LeaveSR ADO_Connection
    EndFunc
    BegSr Dispose Access(*Public) Modifier(*Overrides)
        DclSrParm disposing Type(*Boolean)
        If disposing
            Disconnect  MyDatabase
            Disconnect  MyPrinterDB
            ADO_Connection.Close()
            ADO_Connection.Dispose()
        EndIf
        *Base.Dispose(Disposing)
    EndSr
   BegSr  ExecuteStartupProgram Access(*Protected) Modifier(*Overrides)
     Connect     MyDatabase
     Connect     MyPrinterDB
     DclFld connectStr Type(*String)
 connectStr=string.Format("Driver={{Client Access ODBC Driver (32-bit)}};" + +
                           "System={0};DBQ={1},*USRLIBL;Uid={2};Pwd={3}", +
                           MyDatabase.Server, +
                           MyDatabase.DBName, +
                           MyDatabase.User,   +
                           MyDatabase.Password )
/Error When using MS SQL Server, replace connectStr assignment with "Data
Source={0};Initial Catalog={1};Integrated Security=True"
        ADO_Connection = *New System.Data.Odbc.OdbcConnection( connectStr )
/Error When using MS SQL Server, replace *New System.Data.Odbc.OdbcConnection with
*New System.Data.SqlClient.SqlConnection
```

```
        ADO_Connection.Open()
        CallD       'MyCompany.MyApplication.Db0720'
    EndSr
EndClass
```

Example - Host Variables

The following legacy source example shows three host variables:

```
C/EXEC SQL
C+ DELETE FROM DEPARTMENT
C+  WHERE WDDATE = :ODATE# and WDEMP# = :F1EMP# and
C+        WDORD# = :F1ORD#
C/END-EXEC
```

The following migrated code uses **DBParm**, **DBStrParm**, and **DBScaledParm** to pass the value for the three host variables.

```
ExecuteSQL_Statement ( +
"DELETE FROM DEPARTMENT WHERE WDDATE = @sql_parm_1 and WDEMP# = @sql_parm_2 +
 WDORD = @sql_parm_3;", +
*New  DBParm(DbType.Time, ODATE#), +
*New  DBScaledParm(DbType.Decimal, F1EMP#, %Len(F1EMP#), %DecPos(F1EMP#)), +
*New  DBStrParm(DbType.String, F1ORD#, %Len(F1ORD#) +
)
```

Example - Into Keyword

The following legacy source has the **Into** keyword removed from the Select statement and replaced with blanks. The host variables are saved and used after executing the statements to retrieve their values. With this legacy source code,

```
C/EXEC SQL
C+ Select WDDATE, WDSEQ#, WDEMP# Into :F1Date, :F1Seq#, :F1Emp#
C+ From wfmmst Join wfmdet On wmemp# = wdemp#
C+ Where wdmsc4 <> 'A' and wdetim = 0 and wdord# = :F1Ord# and
C+       wdmsc1 = 'P'
C/END-EXEC
```

This migrated code is generated.

```
SqlQueryResults = ExecuteSQL_Query ( +
"Select WDDATE, WDSEQ#, WDEMP#                          " + +
" From wfmmst Join wfmdet On wmemp# = wdemp#"  + +
" Where wdmsc4 <>  'A' and wdetim = 0 and wdord# = @sql_parm_1 and" + +
"       wdmsc1 = 'P';", +
*New  DBStrParm(DbType.String, F1Ord#, %Len(F1Ord#)) +
)
F1Date = SqlQueryResults["WDDATE"] *AsFld F1Date
F1Seq# = SqlQueryResults["WDSEQ#"] *AsFld F1Seq#
F1Emp# = SqlQueryResults["WDEMP#"] *AsFld F1Emp#
```

Example - Select Expression

Select statements that have an expression instead of a list of columns, such as **Count(*)**, are modified by inserting "as *SqlExprResultColumn*" so that a new column is added to the result table which can be accessed using SqlQueryResults["SqlExprResultColumn"] shown in the following example:

```
C/EXEC SQL
C+ Select Count(*)
C+ Into :#Count
C+ From wfmmst Join wfmdet
C+ On   wmemp# = wdemp#
C+ Where wdmsc4 <> 'A' and wdetim = 0 and wdosst = ' ' and
C+       wdord# = :F1Ord#
C/END-EXEC
```

This migrated code is generated as:

```
SqlQueryResults = ExecuteSQL_Query ( +
"Select Count(*) as SqlExprResultColumn " + +
"              " ++
" From wfmmst Join wfmdet"  + +
" Where wdmsc4 <>  'A' and wdetim = 0 and wdosst = ''  and" + +
"       wdord# = @sql_parm_1;", +
*New  DBStrParm(DbType.String, F1Ord#, %Len(F1Ord#)) +
)
#Count = SqlQueryResults["SqlExprResultColumn"] *AsFld #Count
```

Example - Declared Cursor with Unsupported Qualifiers

With this legacy source:

```
C/EXEC SQL
C+ DECLARE Cursor1
C+ INSENSITIVE     SCROLL CURSOR
C+ WITHOUT HOLD WITH RETURN
C+ FOR
C+ SELECT FLD1, FLD2, FLD3
C+     FROM
C+        TBLA, TBLB
C+     WHERE
C+         :PARAM1 < KEY1 AND
C+         :PARAM2 LIKE "AAAA" AND
C+         :PARAM3 < KEY2
C+     ORDER BY
C+         TBLA DESC
C+     FETCH FIRST 2 ROWS ONLY
C+     OPTIMIZE FOR 2 ROWS
C+     WITH RR
C/END-EXEC
```

This migrated code is generated.

```
/Error Unsupported: "INSENSITIVE" CURSOR qualifier not supported for DECLARE CURSOR.
/Error Unsupported: "WITHOUT HOLD" CURSOR qualifier not supported for DECLARE CURSOR.
/Error Unsupported: "WITH RETURN" CURSOR qualifier not supported for DECLARE CURSOR.
DclFld Cursor1 Type(SqlCursor)
Cursor1 = *New SqlCursor( monarchJob.ADO_Connection, "SELECT FLD1, FLD2, FLD3" + +
"     FROM" + +
"        TBLA, TBLB" + +
"     WHERE" + +
"         @sql_parm_1 < KEY1 AND" + +
"         @sql_parm_2 LIKE 'AAAA' AND" + +
"         @sql_parm_3 < KEY2" + +
"     ORDER BY" + +
"         TBLA DESC" + +
"     FETCH FIRST 2 ROWS ONLY" + +
"     OPTIMIZE FOR 2 ROWS" + +
"     WITH RR;",
*New  DBStrParm(DbType.String, PARAM1, %Len(PARAM1)), +
*New  DBStrParm(DbType.String, PARAM2, %Len(PARAM2)), +
*New  DBStrParm(DbType.String, PARAM3, %Len(PARAM3))  +
```

```
)
Cursor1.ScrollType = SqlCursor.ScrollTypes.Scrollable
```

Example - Prepared Statements

The following segments of legacy code (where "SelectF1" is a *Char len(500)) dynamically assigned to a valid SELECT statement.

```
C/EXEC SQL
C+ PREPARE SelF1 FROM :SelectF1
C/END-EXEC
C* Declare the SQL cursor to hold the data retrieved from the SELECT
C/EXEC SQL
C+ DECLARE SelF1CSR SCROLL CURSOR FOR SELF1
C/END-EXEC
C* Open the SQL cursor.
C/EXEC SQL
C+ OPEN SelF1CSR
C/END-EXEC
C* Process the records in the SQL cursor until the return not = 0
C                    Do        SubfilePage
C*
C* Get the next row from the SQL cursor.
C/EXEC SQL
C+   FETCH NEXT FROM SelF1CSR
C+   INTO :F1DATE, :F1ORD#, :F1SEQ#, :F1EMP#, :F1WFGR, :F1DIST
C/END-EXEC
C*
C                    If        SQLCOD = 0
                          Success do something
C                    Else
                          Failure, exit loop
C                    Leave
C                    EndDo
C/EXEC SQL
C+ CLOSE SelF1CSR
C/END-EXEC
```

Generates this migrated code:

```
/Error: Validation of dynamic SQL statement using host variables delayed until run-
time. Please review migrated code.
DclFld SelF1 Type(SqlPreparedStatement)
SelF1 = *New SqlPreparedStatement( monarchJob.ADO_Connection, SelectF1 )
//* Declare the SQL cursor to hold the data retrieved from the SELECT
DclFld SelF1CSR Type(SqlCursor)
SelF1CSR = *New SqlCursor( monarchJob.ADO_Connection, SELF1 )
SelF1CSR.ScrollType = SqlCursor.ScrollTypes.Scrollable
//* Open the SQL cursor.
SelF1CSR.Open()
//* Process the records in the SQL cursor until the return not = 0
DoToVal(SubfilePage)
//* Get the next row from the SQL cursor.
 If ( SelF1CSR.Fetch( SqlCursor.FetchOrientations.Next ) )
     F1DATE = SelF1CSR.SqlQueryResultsByIndex( 0 ) *AsFld F1DATE
     F1ORD# = SelF1CSR.SqlQueryResultsByIndex( 1 ) *AsFld F1ORD#
     F1SEQ# = SelF1CSR.SqlQueryResultsByIndex( 2 ) *AsFld F1SEQ#
     F1EMP# = SelF1CSR.SqlQueryResultsByIndex( 3 ) *AsFld F1EMP#
     F1WFGR = SelF1CSR.SqlQueryResultsByIndex( 4 ) *AsFld F1WFGR
     F1DIST = SelF1CSR.SqlQueryResultsByIndex( 5 ) *AsFld F1DIST
 EndIf
 SQLCA = SelF1CSR.SQLCA
 If ( SQLCA.SQLCOD = 0 )
       Success do something
 Else
```

```
       Failure, exit loop
 EndIf
EndDo
```

Example - INSERT and INSERT with VALUES Keyword

The following legacy source example shows an **INSERT**:

```
C/EXEC SQL
C+ INSERT INTO PRMPTP
C+ SELECT 'EMPLOYEE' AS PRFLD, DIGITS(WMEMP#) AS PRVALU,
C+        WMNAME AS PRDESC
C+ FROM WFMMST
C+ WHERE WMEMP# <> :F1EMP#
C/END-EXEC
```

Migrates as:

```
ExecuteSQL_Statement ( +
"INSERT INTO PRMPTP" + +
"SELECT 'EMPLOYEE' AS PRFLD, DIGITS(WMEMP#) AS PRVALU," + +
"       WMNAME AS PRDESC" + +
"From WFMMST" + +
"WHERE WMEMP# <>  = @sql_parm_1;", +
*New  DBScaledParm(DbType.Decimal, F1EMP#, %Len(F1EMP#), %DecPos(F1EMP#))
)
```

The following legacy source example shows an **INSERT** with **VALUES** keyword:

```
C/EXEC SQL
C+ INSERT INTO SVSCC
C+ (SCCSTS, SCCDO#, SCCDAT, SCCTIM, SCCOSC, SCCNSC, SCCUSR),
C+  VALUES(' ', :SvOrd#, :Msoedt, :#Time6, :Msostg, :@Stg, :Sdsusr
C/END-EXEC
```

Migrates as:

```
ExecuteSQL_Statement ( +
"INSERT INTO SVSCC" + +
" (SCCSTS, SCCDO# SCCDAT, SCCTIM, SCCOSC, SCCNSC, SCCUSR)" + +
" VALUES(' ', @sql_parm_1, @sql_parm_2, @sql_parm_3, @sql_parm_4, @sql_parm_5,
@sql_parm_6,;" +
*New DBStrParm(DbType.String, System. SvOrd#, %Len(SvOrd#)), +
*New DBScaledParm(DbType.Decimal, Msoedt, %Len(Msoedt), %DecPos(Msoedt)), +
*New DBScaledParm(DbType.Decimal, #Time6, %Len(#Time6), %DecPos(#Time6)), +
*New DBStrParm(DbType.String, Msostg, %Len(Msostg)), +
*New DBStrParm(DbType.String, @Stg, %Len(@Stg)), +
*New DBStrParm(DbType.String, Sdsusr, %Len(Sdsusr))
)
```

Example - SELECT with Host Variables

This legacy source example shows how variables are concatenated to create the SQL select statement.

```
C/EXEC SQL
C+ INSERT INTO WFMDET
C+ ( Select :#TgtDate, :F4EMP#, :F4SEQ#, WDORD#,
C+          WDTIME, WDSTIM, WDETIM, WDSDTE, WDEDTE, WDASWU,
C+          ' ', WDCTIM, WDCLDT, WDCLTM, WDASBY, WDASDT,
```

```
C+  . . .
C+ FROM WFMDET
C+ WHERE WDDATE = :#ODATE and WDEMP# = :F1EMP# and
C+       WDORD# = :F1ORD# and WDSEQ# = :F1SEQ#
C/END-EXEC
```

Migrates as:

```
ExecuteSQL_Statement ( +
"INSERT INTO WFMDET" + +
" (Select " + %Char(#TgtDate) + ", " +%Char(F4EMP#) + ", " + %Char(F4SEQ#) + ",
WDORD#," + +
"          WDTIME, WDSTIM, WDETIM, WDSDTE, WDEDTE, WDASWU," + +
"          ' ', WDCTIM, WDCLDT, WDCLTM, WDASBY, WDASDT," + +
" . . .
" FROM WFMDET" + +
" WHERE WDDATE = @sql_parm_1 and WDEMP# = @sql_parm_2 and" + +
"       WDORD# = @sql_parm_3 and WDSEQ# = @sql_parm_4;" +
*New  DBParm(DbType.Time, #ODATE), +
*New  DBScaledParm(DbType.Decimal, F1EMP#, %Len(F1EMP#), %DecPos(F1EMP#)), +
*New  DBStrParm(DbType.Decimal, F1ORD#, %Len(F1ORD#)), +
*New  DBScaledParm(DbType.String, F1SEQ#, %Len(F1SEQ#), %DecPos(F1SEQ#)), +
)
```

Example - UPDATE

This legacy source example shows the SQL UPDATE command.

```
C/EXEC SQL
C+ Update WFMDET
C+ Set WDSTIM = 0, WDETIM = 0, WDOSST = :@OrderStat
C+ Where WDDATE = :#OrdDate and WDEMP# = :F1EMP# and
C+       WDORD# = :F1ORD# and WDSEQ# = :F1SEQ#
C/END-EXEC
```

Migrates as:

```
ExecuteSQL_Statement ( +
"Update WFMDET" + +
" Set WDSTIM = 0, WDETIM = 0, WDOSST = @sql_parm_1" + +
" WHERE WDDATE = @sql_parm_2 and WDEMP# = @sql_parm_3 and" + +
"       WDORD# = @sql_parm_4 and WDSEQ# = @sql_parm_5;", +
*New  DBStrParm(DbType.String, @OrderStat, %Len(@OrderStat), +
*New  DBParm(DbType.Time, #OrdDate), +
*New  DBScaledParm(DbType.Decimal, F1EMP#, %Len(F1EMP#), %DecPos(F1EMP#)), +
*New  DBStrParm(DbType.Decimal, F1ORD#, %Len(F1ORD#)), +
*New  DBScaledParm(DbType.String, F1SEQ#, %Len(F1SEQ#), %DecPos(F1SEQ#)), +
)
```

Monarch Framework Support

The following table contains a brief recap of the classes to support the migration of embedded SQL.

CLASSES	DESCRIPTION AND MEMBERS OF NOTE
DBParm	Used for constructing SQL parameters where the attributes of the parameter can be determined by its' DbType.
DBStrParm	Used for constructing SQL parameters where the parameter is DbType.String requiring the length (%Len) to be specified.
DBScaledParm	Used for constructing SQL parameters where the parameter is DbType.Decimal requiring the length (%Len) and decimal positions (%DecPos) to be specified.
Program	ExecuteSQL_Query, ExecuteSQL_Statement, ExecuteSQL_QueryVerbatim, and ExecuteSQL_StatementVerbatim, for the execution of SQL commands and SqlQueryResults to contain the results.
SQL_CommunicationsArea	Represents the global SQLCA to trap and report run-time errors for a SQL statement. Main property of note is SQLCOD and SQLSTT used to test the status of the last executed SQL statement.
SqlCursor	Represents a SQL cursor for a multi-row dataset on which the cursor methods operate. It has its own SQLCA and SqlQueryResults properties as well as the SqlQueryResultsByIndex method for result set index access.
SqlPreparedStatement	Represents a prepared executable version of a SQL command statement having its own SQLCA.

Array Translation

RPG arrays go through two basic conversions:

- AVR uses brackets ('[' and ']') to access array elements instead of parenthesis.
- The first element is indexed by *zero*, as opposed to by *one* in legacy RPG.

For example the following legacy code snippet:

```
C     Char10         Lookup    SArrayi(3)      20
```

Is migrated as:

```
Lookup  Table(SArrayi[2])    Source(Char10)    Eq(*IN20)
```

Notice the changes.

- SArrayi(3) became SArrayi[2]
- The parenthesis was replaced by brackets
- The numeric indexed was decremented by one

Note: *If the index is not a constant, but a variable (for example "x"), then the new code would like this: SArrayi[x-1]*

Hexadecimal Constants

Constants of the form X'xxxx' are not valid. However the format H'xxxx' is supported.

Compile-Time Data

Monarch supports RPG arrays with elements initialized at compile-time.

During RPG code parsing, those arrays defined using keywords CTDATA, PERRCD, and ALT are collected and when the new code is generated, the following code is added to load the data into the array elements.

AVR Program Constructor

The first thing the migrated program (class) does, when using compile-time data arrays, is call one subroutine per array to load its data.

The naming convention used for these Monarch generated subroutines is:

```
Load_CompileTime_Table_ArrayName
```

For Example, a program that defines the arrays:

```
  DclArray ERR       Type(*Char )    Len(70 )    Dim(1)
  DclArray TABJG1    Type(*Char )    Len(1 )     Dim(12)
  DclArray TABJG2    Type(*Char )    Len(15 )    Dim(12)
  DclArray TABCT1    Type(*Zoned)    Len(3,0)    Dim(102)
  DclArray TABCT2    Type(*Char )    Len(12 )    Dim(102)
  DclArray LTR       Type(*Char )    Len(1  )    Dim(26)
```

Will include the following Monarch generated calls in the constructor:

```
BegConstructor    Access(*Public)
    Load_CompileTime_Table_ERR()
    Load_CompileTime_Table_TABJG1()
    Load_CompileTime_Table_TABJG2()
    Load_CompileTime_Table_TABCT1()
    Load_CompileTime_Table_TABCT2()
    Load_CompileTime_Table_LTR()
           . . .
EndConstructor
```

Generated compile-time loading subroutines

The implementation of each of those subroutines is generated by the RPG Agent at the end of the program's file (inside a ***region*** named "** Compile-time Data"), as shown in the next figure:

```
/region ** Compile-time Data
   BegSr Load_CompileTime_Table_LTR
        LTR[0] = "A"
        LTR[1] = "B"
        LTR[2] = "C"
        LTR[3] = "D"
        LTR[4] = "E"
        LTR[5] = "F"
        LTR[6] = "G"
           .
        LTR[19] = "T"
        LTR[20] = "U"
        LTR[21] = "V"
        LTR[22] = "W"
        LTR[23] = "X"
        LTR[24] = "Y"
        LTR[25] = "Z"
    EndSr
/endregion
```

Markers Supported as Start of "compile-time data"

The two markers supported by Monarch to indicate the start of the compile-time data are:

Marker	Comments
"** "	Two asterisks in positions 1 and 2 and a blank in position 3. ***Note:*** *The data for the arrays must be specified in the same order in which they are specified in the Definition specifications.*
"**CTDATA *arrayname*"	Two asterisks in positions 1 and 2 and a blank in position 3, plus the name of the array. If the *arrayname* has more than one name (separated with blanks or commas), only the first name is used. ***Note:*** *The data for the array may be specified anywhere in the compile-time data section.*

Other Considerations

Internally-Described Files

Monarch supports Program-described Printer Files and its corresponding EXCEPTs described in detail in Chapter 10. EXCEPTs used to update individual fields are also supported.

Monarch does not currently support Program-described database files. The Migrator needs to obtain externally described files. A product like ASNA ProStart can be used to create database files that accurately define their internal field composition.

As part of the automatic migration process in a future release, Monarch will be able to match up the external field names and types with the one found in the Program I and O specs. Monarch will then emit appropriate renames or aliases to have the Program refer accurately to the external file definition.

There are a few cases when this match-up will not be accomplished. These include:

- Multiple record formats within a single file
- Overlapping fields that are not wholly contained in a parent field
- Two renamed-fields in different files sharing the same name
- Unmatched field name list between the I-Specs and the O-spec

Chapter 9 - Advanced RPG Data Structure Migration

Overlapping Data Structures

Data Structures with Gaps

Overlapping data structures are used to remap areas of memory allowing the program to give multiple meanings to the same bytes. .NET strictly enforces type-safety and dislikes this kind of arbitrary reinterpretation of memory. Remapping is sometimes risky but there are certain common usages that are convenient and valid.

Legacy RPG allows specification of fields within a data structure with explicit positions and length, opening the possibility of gaps in the memory map, for example:

```
D DATEDS          DS            10
D  MM                    1       2  0
D  DD                    4       5  0
D  YY                    7      10  0
```

See how position ***three*** and ***six*** in the above example are not used. The legacy programmer most likely mapped the data structure to an existing Data-Area where those positions are used for other purposes.

Monarch will create fields with unique names and the proper length to guarantee that the relative positions of the fields remains intact.

For example, the generated code for this Data Structure would be:

```
DclDS DATEDS
  DclDsFld MM      Type(*Zoned) Len(2,0)
  DclDsFld MonarchName28 Type(*Char ) Len(1)    // Monarch added filler
  DclDsFld DD      Type(*Zoned) Len(2,0)
  DclDsFld MonarchName29 Type(*Char ) Len(1)    // Monarch added filler
  DclDsFld YY      Type(*Zoned) Len( 4,0 )
```

Giving individual field names to elements of an array

Consider the following data structure declaration:

```
D CUSTSL        E DS                    EXTNAME(CSMASTER)
D   SlsArray             11      82P02 DIM(12)
```

Where CSMASTER file has the following definition:

The field SlsArray *overlaps* the fields: CSSALES01 thru CSSALES12. The intention is to be able to refer to all twelve packed numbers (year sales) as a block or to refer to one of the months sales through a field name, for example CSSALES06. Effectively the desired behavior is to make SlsArray(6) an element of an array and refer to a field name in a data structure.

The RPG Agent will bring the external structure into the migrated code and take advantage of one of Visual RPG commands, namely **DclOverlayGroup** to produce the following migrated code:

```
// DS CUSTSL has overlapping fields
// Data structure "CUSTSL" was externally described, but was changed to internally
described.
DclDS CUSTSL ExtDesc(*Yes)     DBDesc(MyJob.MyDatabase)    FileDesc("*LIBL/CSMASTER")
  DclDsFld CSCUSTNO           Type(*Packed)       Len(9,0)
  DclDsFld CSYEAR             Type(*Zoned)        Len(4,0)
  DclDsFld CSTYPE             Type(*Zoned)        Len(1,0)
  DclDsFld CSSALES01          Type(*Packed)       Len(11,2)
  DclDsFld CSSALES02          Type(*Packed)       Len(11,2)
  DclDsFld CSSALES03          Type(*Packed)       Len(11,2)
  DclDsFld CSSALES04          Type(*Packed)       Len(11,2)
  DclDsFld CSSALES05          Type(*Packed)       Len(11,2)
  DclDsFld CSSALES06          Type(*Packed)       Len(11,2)
  DclDsFld CSSALES07          Type(*Packed)       Len(11,2)
  DclDsFld CSSALES08          Type(*Packed)       Len(11,2)
  DclDsFld CSSALES09          Type(*Packed)       Len(11,2)
  DclDsFld CSSALES10          Type(*Packed)       Len(11,2)
  DclDsFld CSSALES11          Type(*Packed)       Len(11,2)
  DclDsFld CSSALES12          Type(*Packed)       Len(11,2)
  DclOverlayGroup SLSARRAY Dim(12) Type(*Packed) Len(11,2)    StartField(CSSALES01)
```

DclOverlayGroup maps the array (SLSARRAY) to the field elements of the data structure CUSTSL starting with CSSALES01, which is exactly what the legacy RPG logic had intended.

Masquerading a Data Structure as a Large Character Field

Consider the following data structure:

```
D LDA            UDS
D  LDA1                 1     256
D  LDA2               257     512
D  DCLEAR             900     905
D  DCCO#              900     902  0
D  DCCOPY             903     904  0
D  DCHOLD             905     905
```

Given that this structure is the LDA, Monarch needs to *implicitly* read it when the program starts (construction) and write it back when the program ends (dispose).

Even when the data structure does not have overlapping fields, this shows the technique of defining the whole data structure as a large character field.

If we look at the code, Monarch will generate to implement the *implicit* write,

```
BegSr Dispose Access(*Public) Modifier(*Overrides)
DclSrParm disposing Type(*Boolean)
  If( disposing )
  // Implicit write of data-area data structures (Monarch generated)
     DclFld flat_LDA Type(*String)
     DsDump LDA flat_LDA
     DataArea_Out( '*LIBL', '*LDA', flat_LDA, *False)
        .
        .
   EndIf
   *Base.Dispose(Disposing)
EndSr
```

Monarch declares a temporary field called flat_LDA as a ***String** and then uses the Visual RPG command **DsDump** to copy the memory used by the data structure into a **flat** array of *Char to be able to use DataArea_Out to write the structure to the System i Data-Area object.

Partitioning a Data Structure Field into Sub-Fields (Date, Time)

Consider the following data structure:

```
D                  DS
D  DATEU                    1      8  0 INZ
D  UC                       1      2  0
D  UY                       3      4  0
D  UM                       5      6  0
D  UD                       7      8  0
```

Notice how this data structure has no name. It is not intended to be used as a collection of fields, but instead to access pieces of the data structure using the sub-fields.

This is a typical example of the legacy RPG code dealing with dates (or time).

The Date and Time data type did not appear in RPG until the mid-nineties with the advent of RPG ILE. Prior to that, RPG Programmers were forced to use a Character (or Decimal) type to hold a date field, and then subdivide the field into the year, month, and day subfields.

The Data structure above would be migrated as:

```
DclDs
  DclDsFld DATEU    Type(*Char)    Len(8)
  DclDsFld UC       Type(*Char)    Len(2) Overlay(DATEU, 1)
  DclDsFld UY       Type(*Char)    Len(2) Overlay(DATEU, 3)
  DclDsFld UM       Type(*Char)    Len(2) Overlay(DATEU, 5)
  DclDsFld UD       Type(*Char)    Len(2) Overlay(DATEU, 7)
```

Partitioning a Data Structure Field into Sub-Fields (Coded Names)

Another common use of partitioning Data Structures into subfields is when dealing with coded names. In the following example, a variable name $glal# represents the name of an account, where the first six digits are used to store the organization name, the next ten for account number and the last four for the location code.

```
     D                  DS
     D $glal#                  1      20  0
     D  $org                   1       6  0
     D  $glno                  7      16  4
     D  $loc                  17      20  0
```

Again, the data structure is unnamed and the intention is to use the whole account code through $glal# or its parts: $org, $glno and $loc (sharing the same memory).

Monarch produces the equivalent migrated code:

```
DclDs
   DclDsFld $glal#    Type(*Zoned)    Len(20)
   DclDsFld $org      Type(*Zoned)    Len(6,0)     Overlay($glal#,1)
   DclDsFld $glno     Type(*Zoned)    Len(10,4)    Overlay($glal#,7)
   DclDsFld $loc      Type(*Zoned)    Len(4,0)     Overlay($glal#,17)
```

Renaming fields and mapping to arrays

Consider the following array declaration and its corresponding Input specifications:

```
     D MSC              S              8     DIM(5)
     IMSHMSTRR
     I        MRMSC1                             MSC(1)
     I        MRMSC2                             MSC(2)
     I        MRMSC3                             MSC(3)
     I        MRMSC4                             MSC(4)
     I        MRMSC5                             MSC(5)
```

Where the record format of the disk file is:

The C-Spec code will use individual elements of the array MSC, to refer to the fields MRMSC1 and MRMSC5 of the record format MSHMSTRR. For example, in the following Do loop:

```
C                    DO        5         I                 2 0
C     MSC(I)         IFNE      *BLANK
C                    MOVE      MSC(I)    MCACOD
```

Monarchs' goal is to produce an equivalent code, as follows:

```
Do ToVal( 5 )  Index( I )
  If ( MSC[ I-1 ] <> *BLANK )
     Move  MSC[ I-1 ] MCACOD
        .
  EndIf
EndDo
```

This is accomplished by the use of the Visual RPG command **DclAliasGroup**.

```
DclArray MSC        Type(*Char) Len(8) Dim(5)
DclAliasGroup Array(MSC) Flds(+
        MRMSC1, +
        MRMSC2, +
        MRMSC3, +
        MRMSC4, +
        MRMSC5  +
) StartIndex(0)
```

The **DclAliasGroup** command is used to 'alias a **field** name to an **array** element'. This effectively renames fields MRMSC1, MRMSC2, MRMSC3, MRMCS4, and MRMSC5 to MSC(1), MSC(2), MSC(3), MSC(4) and MSC(5), which was the desired result.

Externally-Described DS augmented by new fields

Consider the following Data Structure declaration:

```
IEXT##1    E DSSKUMAST
I          IMONHAND                      IMHAND
I          IMONORD                       IMORD
I          IMCOMMIT                      IMCOMT
I          IMUNITCOST                    IMCOST
I          IMUNITPRIC                    IMPROC
I                               77   80 NEWSKU
```

Where the file SKUMAST is defined by:

The new Data Structure is a combination of the original fields as defined by the file - except **some** are ***renamed*** - augmented by a new field.

Monarch changes the data structure from externally-described to internally-described to show the resulting data structure:

```
DclDS EXT##1
// Data structure "EXT##1" was externally described, but was changed to internally
described.
   DclDsFld IMSKU        Type(*Packed)    Len(5,0)
   DclDsFld IMNAME       Type(*Char)      Len(40 )
   DclDsFld IMHAND       Type(*Packed)    Len(7,0)
   DclDsFld IMORD        Type(*Packed)    Len(7,0)
   DclDsFld IMCOMT       Type(*Packed)    Len(7,0)
   DclDsFld IMCOST       Type(*Packed)    Len(11,4)
   DclDsFld IMPROC       Type(*Packed)    Len(11,4)
   DclDsFld NEWSKU       Type(*Char)      Len(4  )
```

Notice how IMSKU and IMNAME are identical to the original file description but the following were renamed:

Original Field Name	**New Field Name**
IMONHAND	IMHAND
IMONORD	IMORD
IMCOMMIT	IMCOMT
IMUNITCOST	IMCOST
IMUNITPRIC	IMPROC

Migrating Data Structures defined locally in Procedures

RPG ILE allows declaring data structures local to a procedure. ASNA Visual RPG supports only declaration of local fields in procedures (not local data structures). However, AVR 8.1 does support fields declared with a ***type*** that may be the ***type of a data structure***.

That is, given the following declaration of a data structure:

```
DclDs sample
  DclDsFld  a Type( *Zoned ) Len( 7,0 )
  DclDsFld  b Type( *Char  ) Len( 20 )
```

A field may be declared of a Type derived from the data structure, for example:

```
DclFld myFieldAsDS Type(sample_DS) *New( *Dft )
```

The example above instantiates a field called **myFieldAsDS** that may be used to access field members of the same type and name as those declared in the data structure 'sample'. For instance:

```
myFieldAsDs.a = 190.45
myFieldAsDs.b = "This is a label"
```

The use of a field declared as an instance of a Data Structure type provides the same functionality as local data structures in RPG ILE but follow the object oriented (.NET) syntax.

That is, it requires the qualification of the field name that defines it and the use of the "dotted" notation.

Monarch makes use of "*field data-structure types*" to migrate procedure local data structure definitions.

In particular, the RPG Agent:

1. Moves the data structure declarations outside the procedure implementation.
2. Replaces the declaration of the data structure with a declaration of a field of Type (ds name appended with _DS) and "newed".
3. 'Qualifies' the data structure field (members) with the name of the data structure using the "dotted" notion everywhere the data structure is referenced outside the body of the procedure.
4. If there is data structure fields initialized with values, each data structure field is initialized before the C spec migrated code 'begins'.
5. Unnamed procedure local data structures become "L1" through "L99" where L1 - L99 refer to the first local, second local, etc.

> ***Note:*** *There is a slight possibility of a name collision in the event the legacy code defines "L1" ... L99" in a procedure. The compiler will catch the duplicate symbol and the Migrator will have to resolve the conflict.*

The following examples should help clarify the AVR code generated in the migration process. The first example shows the migration of a simple data structure. The second example shows the migrated code when data structure fields have initialization values. The third and fourth examples shown unnamed data structures and the migrated code generated when this occurs.

Example 1: Migration of local DS pmDate in Procedure DayOfTheWeek

```
0196.00     P DayOfTheWeek    B                    Export
0197.00
0198.00     D DayOfTheWeek    PI                   Like(Days)
0199.00     D  pmDate                              Like(Date)
0201.00     D dsDate               DS
0202.00     D  dsCentury                         2S 0
0203.00     D  dsYear                            2S 0
0204.00     D  dsMonth                           2S 0
0205.00     D  dsDay                             2S 0
```

Generates this migrated AVR code:

```
//Monarch warning: Local DS are not supported, the following DS were moved
outside the body of the procedure that defines it.
DclDs DayOfTheWeekdsDateMonarchType FldScope (*Local)
   DclDsFld dsCentury   Type( *Zoned ) Len( 2,0 )
   DclDsFld dsYear      Type( *Zoned ) Len( 2,0 )
   DclDsFld dsMonth     Type( *Zoned ) Len( 2,0 )
   DclDsFld dsDay       Type( *Zoned ) Len( 2,0 )
BegProc DayOfTheWeek Like( Days ) Access ( *Public )
   DclSrParm pmDate   Like( Date )
DclFld dsDate Type(DayOfTheWeekdsDateMonarchType_DS) *New( *Dft )
. . .
. . .
EndProc
```

Example 2: ErrCode DS with DS fields with initialization data.

```
1453.00     P qusCrtUS        B
1454.00
1455.00     D qusCrtUS        PI                  Like(Error)
1465.00     D ErrCode         DS                  Qualified
1466.00     D  BytesIN                       9B 0 Inz(*Zero)
1467.00     D  BytesOut                      9B 0 Inz(*Zero)
1468.00     D  Id                            7A   Inz(*Blanks)
```

Generates this migrated AVR code (note that the comment is shown on two lines for readability):

```
//Monarch warning: Local DS are not supported, the following DS were moved
outside the body of the procedure that defines it.
DclDs qusCrtUSErrCodeMonarchType FldScope (*Local)
   DclDsFld BytesIN   Type( *Binary ) Len( 9,0 )
   DclDsFld BytesOUT  Type( *Zoned )  Len( 9,0 )
   DclDsFld Id        Type( *Zoned )  Len( 7 )

BegProc qusCrtUS  Like( Error ) Access ( *Public )
   DclSrParm pmUserSpace    Like( NameQual ) By( *Reference )
   DclSrParm pmExtAtr   Like( Name400 ) By( *Reference )
   DclSrParm pmInitSize Type( *Binary ) Len( 5,0 )
   DclSrParm pmInitValue Type( *Char ) Len( 1 )
   DclSrParm pmAuth Like( Name400 ) By(*Reference)
   DclSrParm pmText Type( *Char ) Len( 50 )

   DclFld ErrCode Type(qusCrtUSErrCodeMonarchType_DS) *New(*Dft)
// Initialization of Data Structure fields (Monarch generated)
   ErrCode.BytesOut =*Zero
   ErrCode.Id = *Blanks
   ErrCode.BytesIN = *Zero
. . .
EndProc
```

Example 3: Unnamed procedure-local data structures defined in procedure zzr000_67

```
0249.00     Pzzr000_67        b                                   MMDDYY to CYYMMDD
0251.00     D                 pi             7  0
0252.00     D mmddyy                         6  0 const
0253.00
0254.00     D                 ds
0255.00     D date_in                1       6  0
0256.00     D in_mmdd                1       4  0
0257.00     D in_yy                  5       6  0
0258.00
0259.00     D                 ds
0260.00     D date_out               1       7  0
0261.00     D out_c                  1       1  0
0262.00     D out_yy                 2       3  0
0263.00     D out_mmdd               4       7  0
0265.00     C                   eval      date_in  = mmddyy
0266.00     C                   eval      out_mmdd = in_mmdd
0267.00     C                   eval      out_yy   = in_yy
0268.00     C                   if        out_yy < 40
0269.00     C                   eval      out_c  = 1
0270.00     C                   else
0271.00     C                   eval      out_c  = 0
0272.00     C                   endif
0273.00
0274.00     C                   return    date_out
0276.00     P                 e
```

Generates this migrated AVR code (note that the comment is shown on two lines for readability):

```
// Monarch warning: Local DS are not supported, the following DS were
moved outside the body of the procedure that defines it.
DclDs zzr000_67L1MonarchType FldScope(*Local)
   DclDsFld date_in          Type( *Zoned )   Len( 6,0 )
   DclDsFld in_mmdd          Type( *Zoned )   Len( 4,0 ) Overlay(date_in, 1)
   DclDsFld in_yy            Type( *Zoned )   Len( 2,0 ) Overlay(date_in, 5)
DclDs zzr000_67L2MonarchType FldScope(*Local)
   DclDsFld date_out         Type( *Zoned )   Len( 7,0 )
   DclDsFld out_c            Type( *Zoned )   Len( 1,0 ) Overlay(date_out, 1)
   DclDsFld out_yy           Type( *Zoned )   Len( 2,0 ) Overlay(date_out, 2)
   DclDsFld out_mmdd         Type( *Zoned )   Len( 4,0 ) Overlay(date_out, 4)
BegProc zzr000_67 Type( *Char )    Len( 7 ) Access(*Public)
   DclSrParm mmddyy          Type( *Packed )  Len( 6,0 ) Const // MMDDYY
   DclFld L1 Type(zzr000_67L1MonarchType_DS) *New(*Dft)
   DclFld L2 Type(zzr000_67L2MonarchType_DS) *New(*Dft)
   L1.date_in    = mmddyy
   L2.out_mmdd   = L1.in_mmdd
   L2.out_yy     = L1.in_yy
   If( L2.out_yy  < 40 )
       L2.out_c    = 1
   Else
       L2.out_c    = 0
   EndIf
   Return L2.date_out
EndProc
```

Example 4: Unnamed procedure-local data structures defined in procedure zzr000_76

```
0309.00     Pzzr000_76        b                            CYYMMDD  to MMDDYY
0310.00
0311.00     D                 pi               6  0         MMDDYY
0312.00     D cyymmdd                          7  0 const    CYYMMDD
0313.00
0314.00     D                 ds
0315.00     D date_in                 1        7  0
0316.00     D                         1        1  0
0317.00     D in_yy                   2        3  0
0318.00     D in_mmdd                 4        7  0
0319.00
0320.00     D                 ds
0321.00     D date_out                1        6  0
0322.00     D out_mmdd                1        4  0
0323.00     D out_yy                  5        6  0
0324.00
0325.00     C                 eval      date_in  = cyymmdd
0326.00     C                 eval      out_mmdd  = in_mmdd
0327.00     C                 eval      out_yy    =  in_yy
0328.00
0329.00     C                 return    date_out
0330.00
0331.00     P                 e
```

Monarch migrates the code like this,

```
// Monarch warning: Local DS are not supported, the following DS were moved
outside the body of the procedure that defines it.
DclDs zzr000_76L1MonarchType   FldScope(  *Local )
   DclDsFld  date_in      Type( *Zoned )    Len( 7,0 )
   DclDsFld  Filler       Type( *Char  )    Len( 1 )
   DclDsFld  in_yy        Type( *Zoned )    Len( 2,0 ) Overlay(date_in, 2)
   DclDsFld  in_mmdd      Type( *Zoned )    Len( 4,0 ) Overlay(date_in, 4)
```

```
DclDs zzr000_76L2MonarchType   FldScope(  *Local  )
   DclDsFld  date_out     Type( *Zoned )     Len( 6,0 )
   DclDsFld  out_mmdd     Type( *Zoned )     Len( 4,0 ) Overlay(date_out, 1)
   DclDsFld  out_yy       Type( *Zoned )     Len( 2,0 ) Overlay(date_out, 5)
BegProc zzr000_76 Type( *Char )  Len( 6 )  Access( *Public )   // CYYMMDD to MMDDYY
   DclSrParm cyymmdd  Type( *Packed )  Len( 7,0 )  Const      //  CYYMMDD
   DclFld  L1 Type(zzr000_76L1MonarchType_DS)  *New(  *Dft )
   DclFld  L2 Type(zzr000_76L2MonarchType_DS)  *New(  *Dft
   L1.date_in  = cyymmdd
   L2.out_mmdd = L1.in_mmdd
   L2.out_yy   = L1.in_yy
   Return L2.date_out
EndProc
```

Notice in the above two migrated code examples:

1. The unnamed data structures (used for date in and date out fields) were moved outside the body of the procedure. The new names given in the first example were **zzr000_67L1MonarchType** and **zzr000_67L2MonarchType** and **zzr000_76L1MonarchType** and **zzr000_76L2MonarchType** for the second.

2. Inside each procedure, two new fields are declared (in the place where the data structures were defined in ILE). The field names given are **L1** and **L2**. For the Type, the name of the data structures specified in 1 above was used with "_DS" appended at the end. At the end of each declaration, the keyword *New was also generated to instantiate these two fields (and initialized with zeros and blanks).

3. Lastly, everywhere a data structure field (member) was referenced (**date_in**, **in_mmdd**, **in_yy**, **out_mmdd**, etc.) the migrated code 'qualified' the field with the name of the data structure using the "dotted" notion i.e. **L1.date_in**, **L1.in_mmdd**, **L1.in_yy**, **L2.out_mmdd**, etc.

Unsupported Overlapping Data Structures

In general, overlapping data structure fields, which are not wholly contained in a parent field, are flagged as errors that have to be dealt with.

The Migrator will need to change the Data Structure using techniques similar to the ones Monarch employs (explained in this chapter). These cases are so particular that a computer cannot automatically translate without detailed knowledge of the full context of its use.

Chapter 10 - Migrating Print Files

Two techniques are employed by RPG report applications to describe the layout of the report; externally described Printer Files and the use of O-Specs to internally describe the layout.

ASNA Visual RPG supports printing using externally defined files but does not directly support printing using O-Specs. It is then the job of Cocoon to migrate printer O-Specs into externally defined Printer Files. Since creating Printer files requires the ability to name them and select the database to locate them, it is convenient to localize the O-Specs embedded in RPG Programs.

Monarch provides two migration agents to deal with these two techniques.

The Printer O-Spec Agent takes the RPG O-Specs and generates a DataGate Print File, while the Print File Agent takes DDS specifications and creates a DataGate Print File.

Print File Special Considerations

RPG Print files were designed with line printers as targets. Most legacy line printers were limited to mono-spaced fonts (also called fixed-pitch fonts) and a fixed height "line-buffer".

To print narrower characters and accommodate more columns on the same paper kind, or to print wider characters to emphasize a heading, **CPI** (Characters per Inch) was used. The CPI represents the number of characters that can be printed horizontally in one inch of paper.

Typical CPIs used were:

1. 10 CPI, *normal* font width to print 80 characters on a letter paper i.e. 8 inches (excluding left and right margin) x 10 characters per inch = 80 total characters.
2. 16 CPI, *compressed* font width to print 132 characters on a letter paper (8.25 inches of printable area x 16 characters per inch = 132 total characters).
3. 5 CPI, *expanded* font width to print twice the width of a *normal* font.

LPI units were also used to indicate the number of Lines per Inch (vertically) that a line printer could print. The most typical value was six lines per inch for a total of 66 lines per page on an 11 inch height paper size.

When describing positions in RPG, the programmer used a model of a fixed-width and fixed-height for the characters to be printed.

Monarch Migrated applications run on a .NET platforms where a character position is ambiguous at best. Print specifications in .NET are much more granular; printers are no longer line printers but devices that can print in almost any position on the paper.

Typical printer resolution is measured in DPI (dots per inch) horizontally and vertically. Typical values are 600 x 600 on a laser printer.

Monarch needs to compute the dot position for every constant or field, both horizontally and vertically out of each character and line specification.

Furthermore, the width of a *dot* varies slightly from printer to printer; therefore, a better unit of measure would be inches (or centimeters).

In the Legacy world, the width of a printable character was 1 / CPI inches and the height of a character was 1 / LPI inches.

A constant to print at position 14, line 5 using a 10 CPI font, would print:

14 x (1 / 10) = 1.4 inches horizontally

5 x (1 / 6) = 0.83 inches vertically

During migration, Monarch needs to convert positions and lines to inches (or centimeters). Instead of using 0.1 and 0.16 as fixed values, Monarch allows the Migrator to specify conversion factors. This is especially important during the modernization of a report where the Migrator might want to use a variable-pitch Font.

In Monarch terminology, the factor that converts positions to horizontal inches is called "Average char width"; while the factor that converts line counts to vertical inches is called "Line Height".

Fonts in .NET use the typographic measurement *Point,* which is equal to 1/72 inch and measures the Font height. The width of a character depends on the design of the Font family where 'M' is the widest letter and 'i' the thinnest.

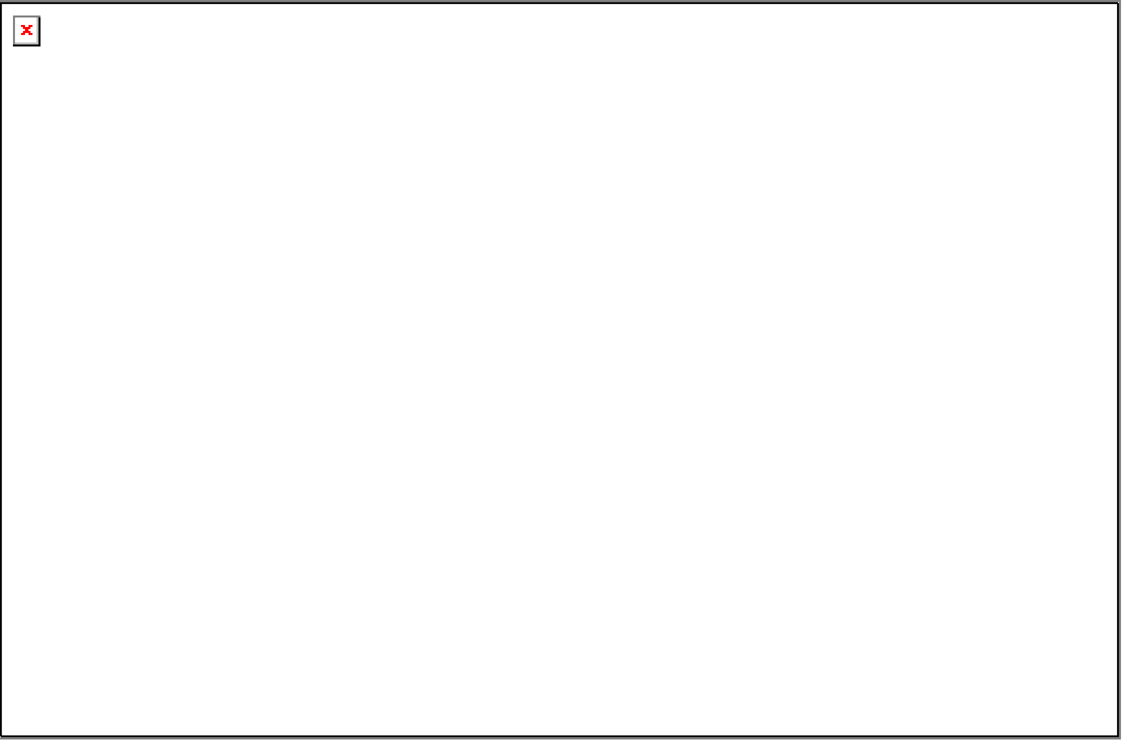
The closest to a mono-spaced *Normal* legacy line-printer font in .NET is "Courier New 11.5 points".

Print File Directives

As part of the migration effort, the Print File **Directives** can be established for the individual Printer and Printer O-Specs Files if different from the default directives outlined in chapter 2 concerning margins, fonts, orientation, and type of paper.

> ***Note:*** *There must be a valid printer driver installed in order for Cocoon to validate these directives as acceptable for the defined printer and to calculate point size, average character width, and line height for the specified valid font. This also includes the Margins, Orientation, and Paper Kind.*

In the figure below on the left hand side under "*Legacy design*", the printer metrics shown corresponds to the originally defined printer file. The "*Migration parameters*" are on the right-hand side. By checking the "*Use default directives*" checkbox, the .Net Print File defaults previously established as .NET Print File Defaults in Chapter 2 will be used as the directives.



Migrating Externally Described Printer Files

The Print File agent takes the externally described DDS print file specifications as input to create a DataGate Print File. This Printer File Agent targets the DataGate Printer File facilities that consist of:

- Print Files
- Print File Designer
- Data aware controls
- XML Spooling
- Render engine

Migrating Printer O-Specs

Output specifications (O-Specs) are used in a RPG Program for two different purposes:

1. As an option to externally described file specification to control when the data is to be written or to specify selective fields that are to be written.
2. To describe a Printer file in a RPG Program.

Monarch only supports the second usage of O-Specs: as Program-Described Printer files. Monarch refers to this type of O-Spec usage as "*Printer* O-Specs".

In addition to the Output specification line, the following two types of specification lines are also associated with Printer O-Specs:

- File Description Specification (F-Spec) with a device entry of type "PRINTER".
- Calculation Specification (C-Spec) lines using the EXCEPT operation code.

About Migrating Printer O-Specs

In order to facilitate the Migration of Printer O-Specs, Monarch includes the concept of a Printer O-Spec object type (not existing on the System i). The Printer O-Spec object type is a reference to the legacy source lines of a Program that defines the Printer file.

The two types of specifications that *define* the Printer O-Spec are:

1. The F-Spec defining the Printer file
2. The O-Spec lines that define the Formats (with its field types and positions) that may be later written (*excepted*) to produce the desired output on paper.

The third type of specification - as mentioned before - the C-Spec with EXCEPT op-codes does not define the Printer file, but rather is part of the program logic to trigger specified formats to be sent to the printer.

Given that the Monarch Printer O-Spec artifact object type does not exist on the System i, but is a new Monarch concept, the collection and object discovery process does not automatically create these new objects on the Gallery.

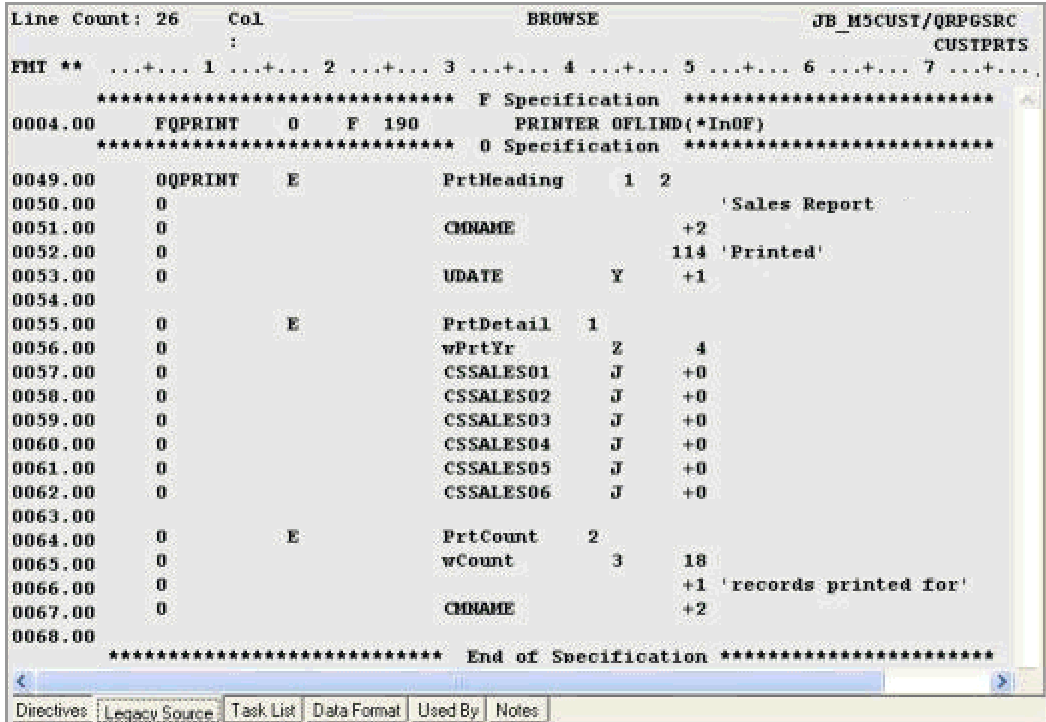
In order for the Printer O-Spec objects to be added to a Gallery or GamePlan, the Source Mappings should be setup correctly so that the entire legacy source for all programs in the Gallery or GamePlan is accessible as discussed in Chapter 2 - Dealing with the Source.

The Printer O-Spec definition depends on the legacy source of the program that describes and uses it. The dependency is so tight, that if the legacy source for the program changes, the Printer O-Spec artifacts needs to be re-discovered.

To run the process to create Printer O-Spec objects, select a Gallery or GamePlan from the solution explorer window and select "**Discover RPG Source Artifacts**" then "**All**" or "**Printer O-Specs**" from the solution explorer context-sensitive menu. Refer to Discovering Artifacts in Chapter 2 for more information on the actual discovery process.

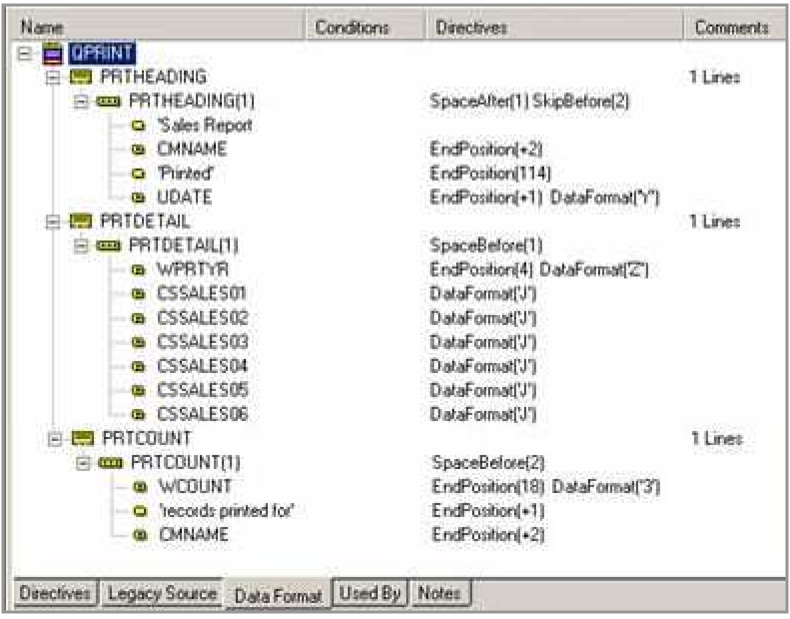
After the discovery process runs, Printer O-Specs can be found in the **Printer O-Specs artifact sub-folder** under the objects in which they are referred. The name of each object is the same as the program that defines it. For every Printer O-Spec, there are several views available.

The **Legacy Source** view displays the source code defining the O-spec. As can be seen in the figure below, "CUSTPRTS" is defined by a printer file named QPRINT that uses indicator *INOF for page overflow logic and defines three record formats: PrtHeading, PrtDetail, and PrtCount.



The layout of each format is described by the O-Specs in a textual format and consists of one or more O-Spec record lines and several O-Spec field lines. As you can see in the Legacy Source above, attempting to describe a rich set of layout instructions on a constrained format results on a hard visualize listing.

In an effort to improve the analysis of the Printer O-Spec, Monarch provides a hierarchical view, with collapsible elements, in the Data Format tab. This view, shown below, describes the same printer file as the listing above.



The Printer name is the root node of the tree ("QPRINT"), followed by all the record formats defined in the file ("PRTHEADING", "PRTDETAIL", "PRTCOUNT").

Each record format may contain zero or more lines. A zero-line record format is typically used to fetch overflow and skip to the next page starting at a certain vertical position. If the record format does not contain any comments in the code, then Monarch will display the number of lines that follow (in this case it shows "1 Line" for "PRTHEADING").

Monarch shows each line of a record format as child nodes. The name of the node is formed by the name of the record format (to which it belongs) with an index appended in parenthesis. For example, the record format "PRTDETAIL" is composed of one line - PRTDETAIL(1). If multiple line were defined for record format "PRTDETAIL", then Monarch would list them as PRTDETAIL(2), PRTDETAIL(3), etc.

The fact that a record format is composed of multiple lines ***does not*** necessarily mean that the record will print that many lines in a report. Each line may be conditioned and/or may skip printer rows before or after printing so the actual number of print lines will depend on the directives associated with each line.

The conditions as well as line directives are shown under the column "Directives". As you can see in the Figure above, none of the record formats for the "QPRINT" print file are conditioned. That means that regardless of the state of the indicators, when the C-Spec executes an EXCEPT, the format will print all its lines.

In addition, by inspecting the Data Format above, we can identify the following directives:

1. Before printing the first line of "PRTHEADING", the printer should advance two full rows.
2. Then it prints the constant 'Sales Report For Customer'.
3. At the end of the constant, the printer should leave two blank spaces and print the contents of field CMNAME.
4. Then, it should print the constant 'Printed' so that the last character ('d') is printed on position 114.
5. One blank is printed at the end of the constant 'Printed', followed by the date using date format "Y".
6. Finally, after printing the one-line record format "PRTHEADING", the printer should advance the printing head, such that one blank row of spacing is left after this format.

Monarch-Generated Record and Field Names

Record Formats without a Name

RPG allows naming exception records so that records with the same name can be grouped together. This name, referred to in RPG as the EXCEPT name, is optional on System i RPG but is required for Monarch to be able to create a DataGate Print file record format.

In RPG:

- E type - those that you can EXCEPT and may or may not have a name
- H type - headers 'printed' at cycle detail-time (usually fetching printer overflow)
- T type - totals 'printed' at cycle total-time

EXCEPT:

- With a name - executes only the 'named' E record
- Without a name - executes all 'unnamed' E records
- For Printer files: means to Write the record
- For Disk files: means to Add, Delete, or Update the record
- For Workstation files: means to Add, Delete, or Update the record of the subfile

In Monarch:

During the process of discovery, Monarch will create unique names for any unnamed records as follows:

- H type = *filename*_DETH*niif*
- D type = *filename*_DET*niif*
- T type = *filename*_TOTAL*niif*
- E type = "CALC_TIME_UNNAMED"

Where *n* is a sequence number starting with the **second** one for the type; *ii* in the indicator used to condition the line; and *f* indicates the line is conditioned by fetch for overflow.

Conditioned Field lines

Whenever a field line is conditioned by indicators (including the overflow indicator), Monarch Printer O-Spec Agent will create unique names by combining the name of the field line (which in turn is made out of the name of the record and an index) and the list of conditioned indicators (including negation).

For instance, if a DETAIL record has two condition lines, where the first line prints only when *IN32 is *Off and *IN29 is *Off; the second line prints only if *IN29 is *Off and *IN17 is *On.

Monarch's Printer O-Spec Agent will split record DETAIL as **DETAIL1N32N29** and **DETAIL2N2917** respectively. The IF and the AND are implied within the name created. In the situation where an OR is used, it is represented in the record name as an underscore ("_"). For example, if the condition for the first detail line was *IN31 is *Off **or** *IN32 is *On; the name created would be **DETAIL1N31_32**.

Remember that the migration of the O Specifications into a DataGate Print File happens before the application is compiled and before the application runs. The concept of indicator state is a *runtime* condition; therefore, Monarch needs to prepare the lines in advance to the application execution.

By turning the conditioned lines into record formats on its own, the RPG Agent can then add logic for the first example like the following:

```
ExSr Write_DETAIL
...
BegSr Write_DETAIL
   If ( *Not *IN32 *And *Not *IN29 )
      Write DETAIL1N32N29
  EndIf
   If ( *Not *IN29 *And *IN17 )
      Write DETAIL2N2917
  EndIf
EndSr
```

This effectively preserves the application logic where the indicators may be tested when the application executes.

Lastly, if fetch for overflow is indicated, an "F" is appended to the name. For example, this legacy source:

```
0303.00     O          EF   77       CustDtls1   2
0304.00     O                        wCstDtlNme
```

Creates this migrated code:

```
BegSr Write_CustDtls1
   If ( *IN77 )
      ExSr *FetchOverflow
      Write CustDtles177F
   EndIf
EndSr
```

See Cycle Support in Chapter 8 for information and examples for a print file within the RPG cycle.

.Net Printfile Field Name Changes

Net Print files are stored by ASNA DataGate as a special multi-format physical file. There are three limitations:

1. Field names are limited to 31 characters.
2. Fields have unique names in a record-format.
3. Array elements for field names are not supported.

How RPG Agent Overcomes these Limitations

1. Only those fields that appear more than once in a particular record format of the Printfile are renamed.
2. See "Rename naming convention 4.0" below for how unique names are created.
3. Both array renames and field renames are implemented as ***Alias*** with a simplified naming convention.

   DclAlias Name(DCT_@1______) Exp(DCT[0])

   DclAlias Name(STKWIP_1____) Exp(STKWIP)

Rename Naming Conventions 4.0:

Names (including array elements) are generated using the following logic:

- For an array element, the new name is created by replacing "(" or "," with an underscore ("_") then appending "@" and the sub-index.

  For example, CLS(9) or CLS, 9 produces a new name of CLS_@9.

- If a name appears more than once in a particular record format, the multiple instances, **starting with the second**, will have an underscore ("_") and sequence number appended to the new name.

  For example, if CSL(9) is used twice, the first becomes CLS_@9 and the second becomes CLS_@9_1, a third instance would become CLS_@9_2, etc.

- The last step is to make sure the new name does not clash with any user-defined names possible in RPG. For that purpose, the length of the new name is increased to exceed the 10-character maximum field name length in RPG (if needed). This is accomplished by appending underscores ("_") to the end of the name to create a field name of 11 characters.

  For example, if the new name is CLS_@9_1 (length is 8), 3 underscore characters would be appended to the end of the new name creating CLS_@9_1___ (length is now 11).

RPG Agent Handling of Printer O-Specs

Substitution of Except for Write

The migrated application will deal with the legacy Printer O-Specs just like any other file. When a particular *EXCEPT name* (now record format) needs to be printed, the record format is written.

All the C-Specs that used to execute the EXCEPT operation are converted by the RPG Agent to a record WRITE. Additionally, whenever an EXCEPT line is migrated by the Printer O-Spec Agent into a conditioned record format (see *conditioned field lines* above), the EXCEPT operation is converted to a subroutine call to a Monarch generated method using the naming convention:

Write_*Except_Name*, for example:

```
EXCEPT     DETAIL
```

Becomes:

```
ExSr Write_DETAIL
```

Handling Page Overflow

The Overflow Indicator

Printer O-Specs may use the overflow indicator to trigger writing certain record formats as headings to a new page. RPG provides the special indicator *INOF for such purpose (although any of the numbered indicators may also be used).

Additionally, the AVR language construct: **DclPrintFile** includes a keyword that accepts an indicator field to be used for Overflow detection. Monarch makes use of such construct to implement handling of legacy page overflow.

For example, the following lines are generated by the RPG Agent for a program that prints using O-Specs with overflow handling logic:

```
DclPrintFile QPRINT DB(MyJob.MyPrinterDB) File("PAYROLL\PAY080") ImpOpen(*No)
OverflowInd(*InOF)
```

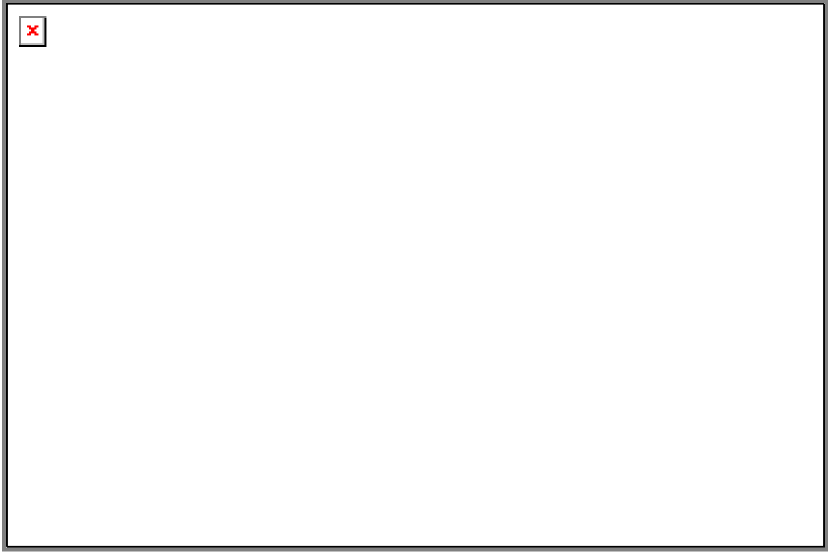
Fetching Overflow versus Being Conditioned on Overflow

DataGate Print file record formats (formerly *EXCEPT names*), may "fetch" for overflow. In Legacy code, fetching for overflow was indicated by an 'F' on position 16, on an O-Spec 'E' type line:

```
O        EF              REPORT              1
```

Fetching for Overflow means that, *before* printing the record format, the Overflow Indicator needs to be checked, and if set, ***all*** the lines that are conditioned on overflow need to be printed.

For example, consider the Printer O-Spec represented by the Figure below.



The Printer O-Spec in the Figure above shows printer file QPRINT with five record formats: BANNER, HEADNG, REPORT, EMPTOT, and TOTLIN. Notice how HEADNG lines one through seven are "*conditioned*" on Overflow. Also notice how record format REPORT includes the directive FetchOverflow(*True).

The logic reflects the fact that the overflow indicator needs to be checked *before* the REPORT lines are printed, and if set, any lines conditioned on overflow (HEADNG lines) need to be printed.

Perhaps it is easier to understand the logic by looking at the Monarch generated code. The EXCEPT REPORT is migrated as:

```
ExSr Write_REPORT
```

where, Write_REPORT is a method implemented as:

```
BegSr Write_REPORT
   ExSr *FetchOverflow
   Write REPORT
EndSr
```

The result is that every time the ExSr Write_REPORT is executed, the value of *InOF is checked (by the *FetchOverflow command). If *On, meaning that the *next* line *would* print inside the page overflow area, the application skips to a line in the next page, prints the heading lines conditioned by overflow, and then continues printing the rest of the report.

Note that the value of *InOF is changed by the AVR runtime *after* writing to ***any*** DataGate print file format. The page overflow area is defined by the Printer-O Spec overflow value set in its directives.

This page is intentionally left blank.

Chapter 11 - Migrating Message Files

Message File Directives

The following figure shows the Directives page for a Message Files showing the location for the migrated files.

The migrated Message File becomes an **XML file**. The following figure shows an example:

Converting XML Message Files to Logical Files

A utility can be used to convert the XML Message Files into logical files.

Install Directory

When the GamePlan is migrated, a **Web.config** file is created in the "<solutionName>web" **Monarch** folder containing the location of the Message Files for the application. This is the location specified in the **Message File Directives** and is the directory in which the Message Files must be installed.

When an application using a message file runs, it will look for those Message Files in the location as shown in the sample Web.config file below.

```
<configuration>
   <appSettings>
       <add key="MonarchMessageFileFolder"
            value="C:\Inetpub\wwwroot\MessageFiles\"/>
   </appSettings>
   <system.web>
  ...
</configuration>
```

If the WebJob.CurrentJob.MessageFileFolder is not null, the *value* entry is considered an absolute path to the folder containing the Message Files.

Chapter 12 - Migrating Data

Data can remain on the System i or it can be moved to a new database platform like Microsoft SQL Server™. Refer to Chapter 15 for additional considerations.

Using Cocoon Data Migration Agents

The migration of Database files is accomplished independent of program migration using DataAgent .exe files. These agents will execute in the background freeing the Migrator to concentrate on program migration. The Data File and Disk File Agents handle the migration of the data and format respectively and the Migrator can periodically check on the status of the data migration effort using the Data Agent Viewer tab in Cocoon.

The **Data Agent** takes System i files and migrates them to DataGate compatible database files, creating a worker process message file. Directives are set that control the number of worker processes, the creation of libraries and files if non-existent, preservation of reference fields, and also target Microsoft SQL servers versus DataGate SQL Servers.

The **Data Agent Result Viewer** provides the ability to view the worker process message file created by the Data Agent. You can view just the errors, the general log, or the combination. The true functionality of the **Result Viewer** is the ability to view the worker process messages regardless of the conditions under which the Data Agent was executed.

In addition, a separate help topic (DataAgent.chm) describes how to use the Data Agent as standalone utilities for the migration of database files.

The following outlines the basics for handling the creation of a Data Only GamePlan and data migration:

In the Gallery:

1. The Exhibits **DB Files Graph** is provided to give a visual representation of the Data File relationships to assist in the decisions for the files to be included in a data only GamePlan.
2. Set the **Directives** for each database file as needed.
3. From the DB Files Graph, select **Create the GamePlan**.

In the GamePlan:

1. Set the Data Migration Directives (analysis and migration) in the Database Files folder **Data Agent Viewer** tab. This sets the basic directives for all database files in the GamePlan. 'Checked' will be the default for Data Only GamePlans. For other GamePlans, 'unchecked' is the default. Run the "**Check all files for Migration**" menu option to set the checkbox to "Migrate" for all database files in the folder.
2. Set the **Data Migration Directives** for individual database files if needed to set directives different from the basic defaults set for all files in the previous step.
3. Use the *Analyze formats* to include problems that may be encountered with the format of a file that may be impacted by the SQL Server used.
4. **Migrate** the GamePlan.

5. Resolve any issues using the Data Agent Viewer tab **ERRORS panel** to find those files that may need your attention.
6. Use the Data Agent Viewer tab **LOG panel** to monitor the migration progress.

Establishing Directives

The directive for an individual file involves these three basic checkboxes, which are self-explanatory.

The Data Migration Configuration directives provide the data agents with the information needed to migrate all of the data files in the GamePlan Database Files folder that are marked for migration.

Target database name is where the files are to be created.

Batch process – Max. Worker Processes is the maximum number of worker processes initiated to process the database files.

Target is Microsoft SQL Server can be selected to create databases for Microsoft SQL Server. Otherwise, the target is DB2/400 compatible database.

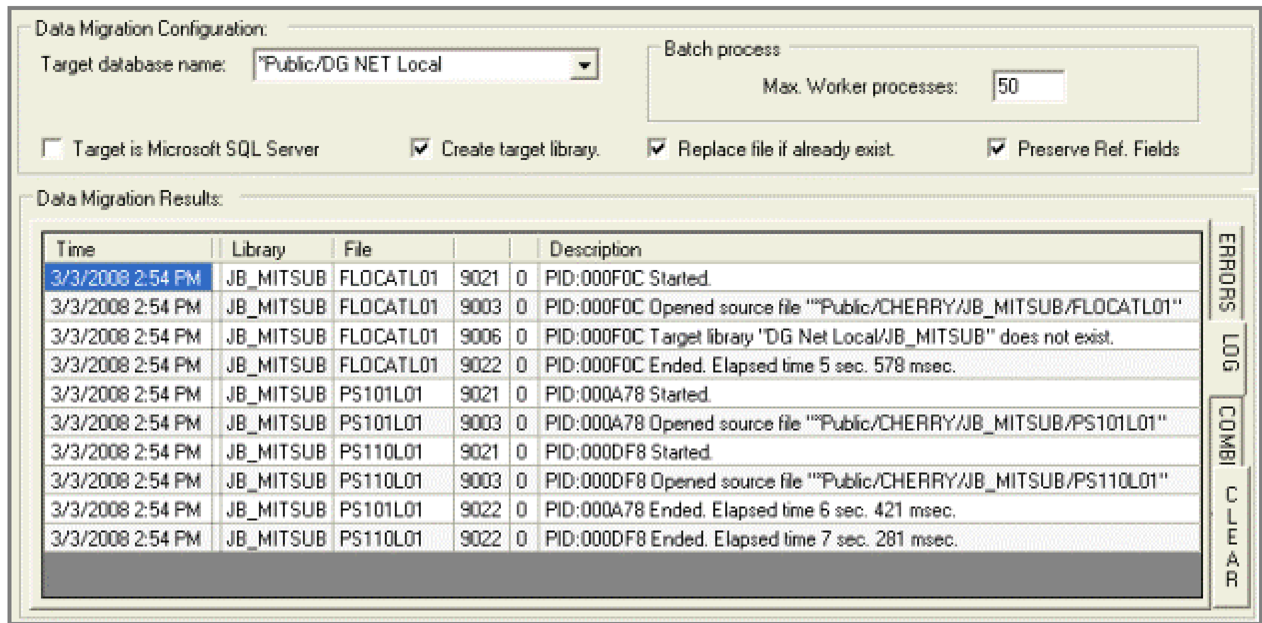
Create target library can be selected if the utility is to create the libraries if they do not exist.

Replace file if it already exist can be selected if the utility is to replace files if they exist.

Preserve Ref. Fields can be selected to preserve the external field references.

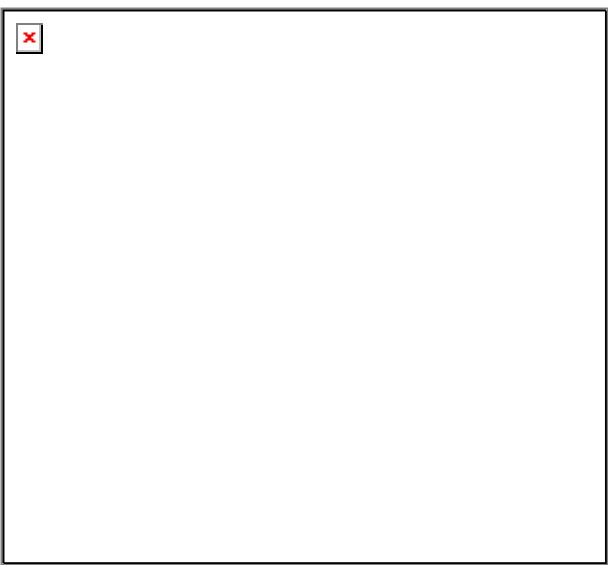
Viewing Data Migration Results

The migration results are contained in three panels - ERRORS, LOG, and COMBINED. The ERROR panel contains information on error messages. The LOG panel contains processing message requiring no action. The COMBINED panel shows both the errors and processing messages in the order of occurrence. Press the CLEAR button to clear the panel contents. An example of a Data Agent Viewer tab follows:



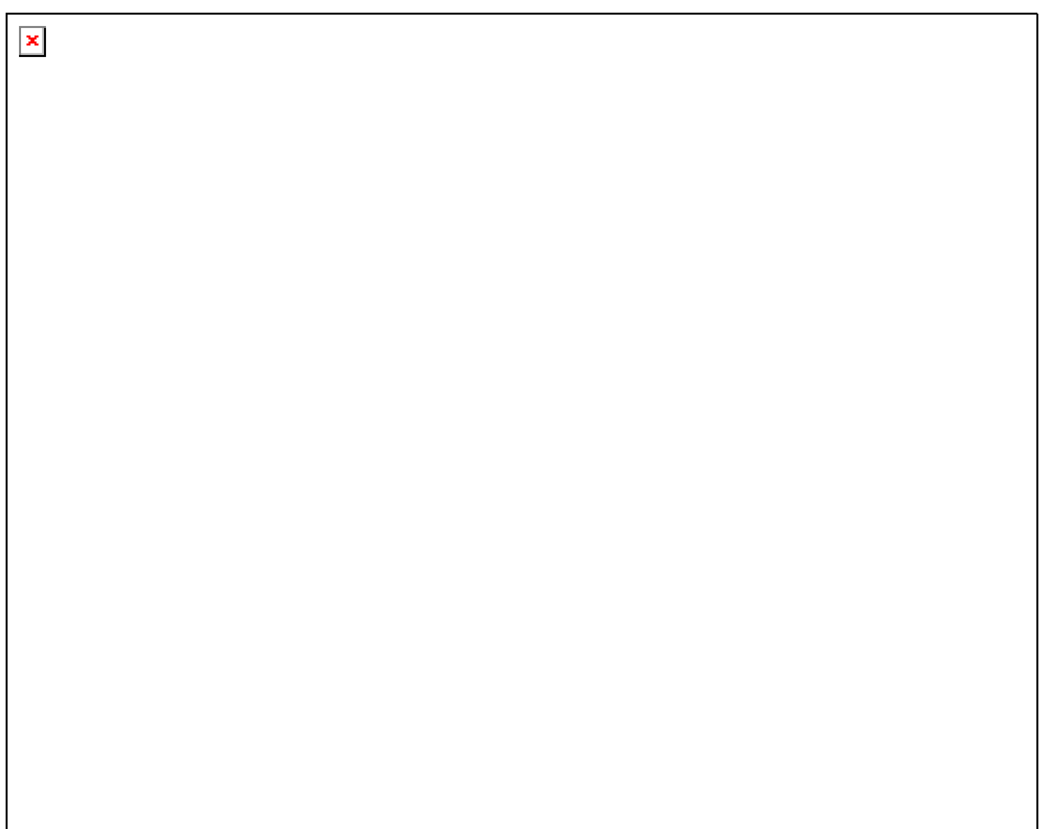
Using DataGate Database Manager

You can also use DataGate Database Manager to copy data, logical files, and an entire or partial library. This section will touch briefly on the three Tool menu option options, **Copy Data**, **Copy Logical File**, and **Copy Library**. At all times, refer to the help topics within DataGate Database Manager for the most current information and other options for copying data and files.



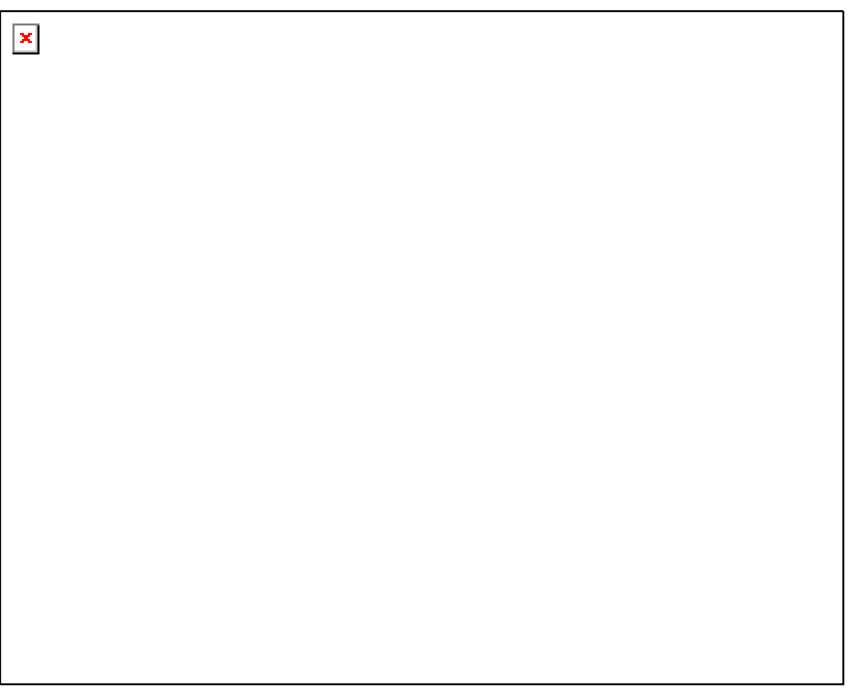
Copying Data

Using the **Tools - Copy Data** option of DataGate Database Manager, the **Copy Data** dialog displays to copy data to the specified location. Notice the additional options to further direct the copy operation i.e. the number of records, order, or even a range.



Copying a Logical File

Using the **Tools - Copy Logical File** option of DataGate Database Manager, the **Copy Logical File** dialog allows data to be copied to the specified location. Notice the additional options to further direct the copy operation i.e. whether to copy the file definitions, and whether to replace the logical and base files if they exist.



Copying a Complete Library

Using the **Tools - Copy Library** option of DataGate Database Manager, the **Copy Library** dialog displays allowing for the entry of the database library to be copied and the target location. Notice the additional checkbox options to direct the types of files to be copied in the operation.

Considerations for DataGate SQL Server

When dealing with applications using multi-member files under SQL Server, several considerations apply.

ASNA DataGate® for Microsoft® SQL Server™ (DSS) does not implement the concept of a multi-member file. When migrating an application that makes use of a multi-member file, it is necessary to modify the application to use multiple files instead of multiple members. Instead of using the add member methods, one must use the copy file methods.

Under DSS, each file is implemented by a table or view depending on whether the file is physical or logical. On the System i, a physical file has several attributes such as the record format definition, which is an order collection of fields, an optional key set at zero or more member that are the actual containers of data records. By contrast, a SQL Server table also defines a collection of columns and contains a set of rows, but the concept of data members is nonexistent.

For illustration purposes, let us assume that the application uses a file called Sales with one member for each month of the year labeled January through December. Furthermore, at the beginning of each month, the application adds a new member to the file where the sales for the month are to be accumulated.

One approach is to create a file with no data that will serve as the model for all the files that will contain the actual data taking the place of the multiple members. Let's call the model file Sales__Model. For each member, a copy of the model file is created using the member name as a suffix to make the name unique. The application uses the appropriate suffixed file to store data, relegating the model file only as a template for the record format and key description. In our example, at the start of a new month, say September, we would copy Sales__Model to Sales_September. Notice that by using a double underscore for the Model file and a single underscore to separate the file name from the suffix (member name) all related files list together under database managers.

If there are logical members associated with the application, a similar approach must be followed taking into account that the base file will have to be modified to accommodate the corresponding new combined file_member name. Also, notice that the longer concatenated name will most likely exceed the 10-character limit imposed by OS/400 but allowed under DataGate for SQL Server.

It is important to use DataGate facilities to create the suffixed files because a physical file is more than just a plain table and a logical file is more than just a view. Special consideration has to be given to the key that ends up being an index on the table, and there are extended properties that have to be added to the table/view to preserve other attributes like column headings and text.

Differences between DataGate, DataGate /400 .NET, and DataGate for SQL Server for .NET

Object Considerations

Item	DG/400	DSS for .NET	DataGate
Library & file name length	10 characters	31 characters	31 characters
Members per file	0 → *NoMax	Exactly 1	0 → *NoMax
File types	Physical Simple logical Join logical Multiformat logical Print	Physical SqlLogical Simple logical Join logical Print	Physical Simple logical Join logical Multiformat logical Print
Max record length	32,000 bytes	8,060 bytes (Not counting Text and Image fields that are not accessible yet by DSS .NET).	32,000 bytes
Max number of records per member		2,147,483,646	2,147,483,646*
Library implemented as:	Library	Database	Illusion
Object text (description)	49 characters	49 characters	49 characters
Stored Procedures	Any AS/400 language	Programmed in SQL-Transact	None
Triggers	Any AS/400 language	Programmed in SQL-Transact	None
Field Reference File (FRF)	A physical file can refer to any number of FRF, which are any physical file in any library. However, DG/400 will report only those coming from the file stated in the DDS REF keyword.	Refers to the collection of 'User Defined Data Types', which is one per Database (i.e.: Library). This collection is surfaced via the special file '*FieldRef' which is the ***ONLY*** file usable as a FRF.	A physical file can refer to only ***ONE*** FRF, which can be any physical file in any library.

** **NOTE**: **For Max number of records per member**, DataGate for Windows and Desktop Servers Release 7.2, Version 7.255 and higher support member/file sizes limit is 16 **exabytes** (2^4 * 2^60).*

Index (Keys) Considerations

Item	DG/400	DSS .NET	DataGate
Indexed logical files per physical file		249	*NoMax
Imposing 'uniqueness' via select/omit rules in logical files	Supported	Not directly supported. See **work-around** note below.	Supported
Logical field used as a key field must be based on a physical field with the same name	No	Yes. Notice that this eliminates the possibility of using Renamed, Concatenated and Sub-stringed fields as keys.	No
Maximum number of key fields per key		16	250
Maximum length of key in bytes	2,000	900	250

Work-around:

As a work-around to support "Imposing 'uniqueness' via select/omit rules in logical files":

- Change the logical file definition to allow duplicate keys.
- Create the logical file.
- Alter the SQL view with the "SCHEMABINDING" attribute.
- Add a Unique, Clustered Index to the SQL view (logical file).
- Make sure the "ARITHABORT" SET option is on/true for the database.

Data Access Considerations

Item	DG/400	DSS .NET	DataGate
Arrival Access	Relative Record Number is used for Sequential and Random access.	Only Consecutive access is supported, but there is no guaranteed order of retrieval unless the file is indexed. The only random operation allowed is SetLL and this is only when used with *Start and *End. No other kind of seeking (SetGT, CHAIN) is allowed.	Relative Record Number is used for Sequential and Random access.
Format Name (see Note below)	Given by file creator.	**Always 'R' followed by File Name.** Note to AVR Users: The Format can be renamed in the DclDiskFile, using the RNMFMT keyword by providing a new name, it is not necessary to provide the existing Name in the RNMFMT. This allows the creation of single-source apps that can compile against DG/400 and DSS .NET.	Given by File creator.
Open Query File	Implemented with OpenQry.	Select expression is used as the WHERE clause of a SELECT. The key field list is used as the ORDER BY clause. The select expression is passed directly to the SQL analyzer with no interpretation. The expression must follow valid SQL Server syntax. Pay special attention to uses of logical operators. Use 'and' and 'or' not '&' and '|'.	A temporary logical file is created using the select expression as a select/omit expression and the key field list to define the new key.

NOTE: *Multi-format logical files are not supported on SQL Server. The migrated code is normalized for SQL Server especially when I/O commands target single-format record format names instead of the file name but this does* ***not*** *change the application's behavior when accessing files on the System i.*

If the Rename keyword is present in the legacy file description, the migrated RNMFMT keyword will contain the legacy New Format Name. If the Rename keyword is ***not*** *present in the legacy file description, the migrated RNMFMT keyword will contain the files Externally Described Record Format name.*

Locking Considerations

Record Locking

DG/400

DB2/400 determines the type and duration of records locks depending on how the file was opened.

For read-only files, when a record is read, there is no lock requested on it, and if some other application has the record lock, the reading application does not block on the lock, that is, the record is read in spite of being locked by somebody else.

For files open for update, every time a record is read it is write-locked so that other updating applications cannot read it. The write lock is held until the record is updated or explicitly unlock by the application or when another record is read or positioned to.

DSS .NET

DSS for .NET (using server cursors) determines the locking characteristics based on how the file is opened.

For read-only files, DSS for .NET behaves like DG/400, that is, there are no locks neither placed nor considered on records being read. The behavior of DSS for .NET when the file is opened for update is similar to DG/400 but with two significant differences: updating a record does not release the lock on the record and explicitly unlocking a record causes the 'current record position' to be lost. These differences bear the following considerations.

Item	DG/400	DSS .NET
Unlock Record	Cursor position is unchanged.	The file has no 'current' position after the *Unlock.*
Update Record	The record just updated is released.	The record just updated is kept locked.
**NoLock* option on *Read* operations	Supported but deprecated.	Unsupported. The better way to achieve this is to open the file twice, once for input only and the other for update. Where the read appears with the *NoLock option, the file should be substituted with the one open for input only. By doing this, the application can take advantage of network blocking - yielding better performance.
Range operations	When the end of the range is reached, the file has no 'current' position.	When the end of the range is reached, the file has no 'current' position.
Hit EOF on a ReadE (P)	Lose Record position.	Lose Record position.
Other Operations like SetLL	Unlock Record.	Unlock Record.

Loops involving SetLL/SetGT and Read/ReadE/ReadPE should be re-coded to use the Range operations. The most demanding change is the one requiring segments of code involving CHAIN-UPDATE. Combinations have to be studied and possibly modified.

- If the CHAIN-UPDATE happens in a tight loop, then at the end of the loop an UNLOCK should be issued to release the last record updated. Note however that the record position will be lost after the UNLOCK.
- If the CHAIN-UPDATE is sprinkled throughout the code, then each case has to be closely studied.

Object Locking

Not implemented on DSS .NET.

Field Considerations

ITEM	DG/400	DSS.NET	DataGate
Field Name Length	10 Characters	31 Characters	31 Characters
Supported			
- Char	*CHAR	char	*CHAR
- Packed	*PACKED	decimal	*PACKED
- Zoned	*ZONED	numeric	*ZONED
- Binary	*BINARY	numeric	*BINARY
- Float	*FLOAT	Float(4): float Float(8): real	*FLOAT
- Integer	*INTEGER	Integer(2): smallint Integer(4): int	*INTEGER
- Date	*DATE	ASNA_DSS.DATE datetime: 00:00:00	*DATE
- Time	*TIME	ASNA_DSS.TIME datetime: 1899/12/30	*TIME
- Timestamp	*TIMESTAMP	datetime	*TIMESTAMP
- Hex	*HEX	binary	*HEX
- DBCS	*DBCS	nchar	*DBCS
- Unicode	*DBCS	nchar	*DBCS
- Boolean	*CHAR(1)	Bit	*CHAR(1)
Allows Nulls	Yes	Yes	No
Variable length Field	Char Hex Dbcs	varchar varbinary varnchar	No
Date value range	0001-01-01 to 9999-12-31	Datetime: 1753-01-01 to 9999-12-31 (0001-01-01 maps to 1753-01-01) Smalldatetime: 1900-01-01 to 2079-06-06	00001-01-01 to 9999-12-31
Decimal number storage	Packed: 1 nibble per digit Zoned: 1 byte per digit Binary 1 - 4 digits: 2 bytes Binary 5 - 9 digits: 4 bytes	Decimal / Numeric 1 - 9 digits: 4 + 1 bytes 10 - 19 digits: 8 + 1 bytes 20 - 29 digits: 12 + 1 bytes 30 - 38 digits: 16 +1 bytes	Packed: 1 nibble per digit Zoned: 1 byte per digit Binary 1 - 4 digits: 2 bytes Binary 5 - 9 digits: 4 bytes
Data storage	1 byte per digit / character	Datetime: 8 bytes ASNA_DSS_DATE: 8 bytes SmallDatetime: 4 bytes	1 byte per digit / character
Fields per file		1,024	32,000
Re-typing logical fields	Unrestricted	Logical fields whose type differs from that of the corresponding physical field cannot be updated	Unrestricted
Column Heading Definitions	Up to 3 31 characters	The 3 headings are concatenated into the MS Access CAPTION field	Up to 3 31 characters
Text Description	Up to 49 characters	Up to 49 characters	Up to 49 characters

Native SQL Server field interpretation

Numerics		Date/Time		Char/Other	
Float->	*Float(4)	DateTime->	*Timestamp	Bit->	*Boolean
Real->	*Float (8)	SmallDateTime>	*Timestamp	Char->	*Char
Int->	*Integer (4)			VarChar->	*Char (VarLen)
SmallInt->	*Integer (2)			NChar->	*Unicode
TinyInt->	*Integer (2)			NVarChar->	*Unicode (VarLen)
Decimal->	*Packed			Binary->	*Hex
BigInt->	*Zoned (19,0)			VarBinary->	*Hex (VarLen)
Money->	*Zoned (19,4)			UniqueIdentifier>	*Hex (16)
Numeric->	*Zoned				
SmallMoney>	*Zoned (9,4)				

The types **Image, Text, and NText** are not supported. Fields of these types are hidden from the file definition. You will be able to display the file definition in Database Manager but will not be able to open the file. To ensure future application compatibility, you should not use files containing these field types. Instead, you should create logical files naming the individual fields that your application will manipulate. That way, if in a future release the fields 'appear', your application will not break.

Join Considerations

Item	DG/400	DSS .NET	DataGate
Supports Use Default for Joins by:	DDS Keyword JOINDFLT When a record is not found in the secondary file, logical fields whose base is that file will be populated with the default values specified in the physical file definition.	Creating a Left Outer Join instead of an Inner Join. From SQL Docs: *LEFT JOIN or LEFT OUTER JOIN* *The result set of a left outer join includes all the rows from the left table specified in the LEFT OUTER clause, not just the ones in which the joined columns match. When a row in the left table has no matching rows in the right table, the associated result set row contains null values for all select list columns coming from the right table.*	Yes
Supports 'Join Duplicates By'	DDS Keyword JDUP	Not supported. Duplicate rows in the 'secondary' tables may be returned in random order.	Yes

Calling Programs/Procedures Considerations

Item	DG/400	DSS .NET	DataGate
Maximum Number of Parameters	36	1024	N/A
Maximum Length of Stored Procedure Name	N/A	31	N/A
Parameter Direction	*Input, *Output, *Both	*Input, *Both	N/A

This page is intentionally left blank.

Chapter 13 - Resolving Unsupported Features

How to Migrate Goto from a Subroutine to Mainline Code

An application migrated with ASNA Monarch will require a small amount of work to handle branching explicitly from one member to another.

RPG on the System i allows GOTO from subroutine code to mainline code. Branching explicitly from one member to another is not supported by the .NET Framework. However, a clever employment of the TRY/CATCH block will allow you to retain the old logic.

Given code that might look like this:

```
C*          Mainline code
C   TAG1       TAG
C              EXSR   SUBRA
C   TAG2       TAG
C              EXSR   SUBRB
C   TAG3       TAG
C              EXSR   SUBRC
C*          Subroutines
C   SUBRB      BEGSR
C                  .
C*          Go to TAG2 on some condition
C              GOTO    TAG2
C                  .
C              ENDSR
C*
C   SUBRC      BEGSR
C                  .
C*             Go to TAG3 on some condition
C              GOTO    TAG3
C                  .
C*             Go to TAG1 on some other condition
C              GOTO    TAG1
C                  .
C              ENDSR
```

The Solution starts by creating your own "GOTO" exception class. Wherever you have a GOTO, you can **THROW** your GOTO exception and include the appropriate **TAG** name as one of its members. Your GOTO class would then look like this:

```
BegClass MyGoto   Extends( System.Exception )
  DclFld  Target    Type( *String )
// constructor receives the Tag Name
  BegConstructor                Access( *Public )
     DclSrParm  GotoTagName     Type( *String )
     Target    = GotoTagName     // save Tag Name globally
  EndConstructor
  BegProp  TagName Type( *String )   Access( *Public )
     BegGet
        LeaveSr Target         // Return the tagname
     EndGet
   EndProp
EndClass
```

Now, the original code as migrated and modified to employ your GOTO class would look like the following:

```
// Mainline code
      TRY
        Tag TAG1
        SubrA()
```

```
        Tag TAG2
        SubrB()
        Tag TAG3
        SubrC()
      CATCH HandleGoto Type( MyGoto )
      ENDTRY
// Cannot execute GOTO inside a TRY/CATCH, so we'll do it here
// if there is an instance of MyGoto
      If HandleGoto <> *nothing
         Select
         When HandleGoto.TagName.ToUpper() = "TAG1"
           Goto Tag1
         When HandleGoto.TagName.ToUpper() = "TAG2"
           Goto Tag2
         When HandleGoto.TagName.ToUpper() = "TAG3"
           Goto Tag3
         EndSL
       EndIf
// Subroutines
      BegSr SUBRB
// Go to TAG2 on some condition
        Throw *new( MyGoto( "TAG2" ))   // Goto TAG2
      EndSr
      BegSr SUBRC
// Go to TAG3 on some condition
        Throw *new( MyGoto( "TAG3" ))   // Goto TAG3
// Go to TAG1 on some other condition
        Throw *new( MyGoto( "TAG1" ))   // Goto TAG1
       EndSr
```

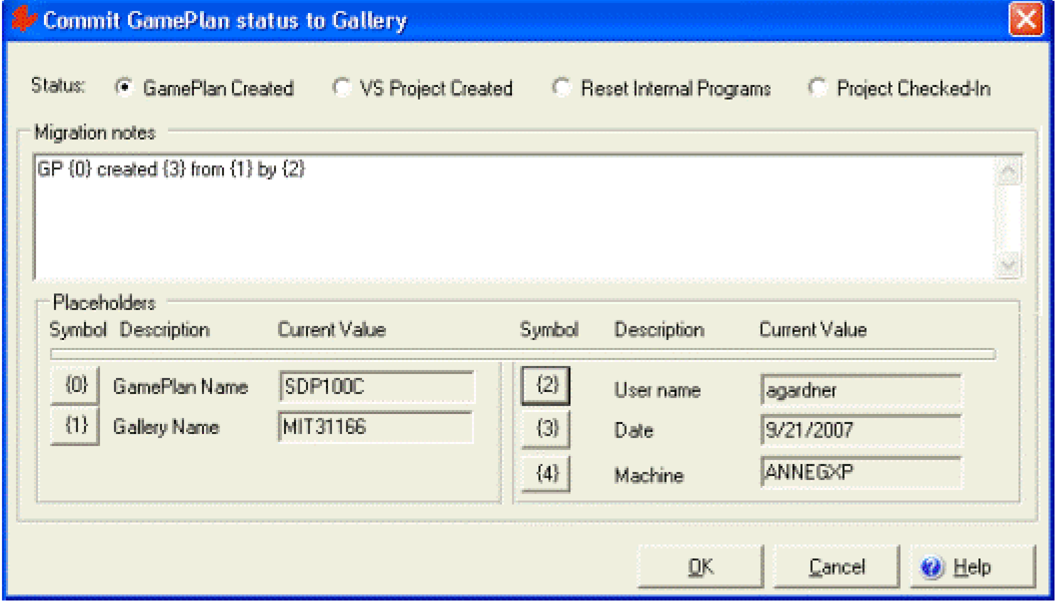
Chapter 14 - Committing GamePlan Status

As part of migration management, the Migrator can commit the GamePlan status back to the Gallery. The normal function takes all "internal" programs in the GamePlan and marks them as "external" in the Gallery. In addition, the namespace and assembly name of the GamePlan are updated on the object directives in the Gallery as needed. A "lock" will then appear in the Gallery for all GamePlan objects indicating its "external" status.

Commit GamePlan Status to Gallery Dialog

The **Commit GamePlan status** option from the solution explorer context-sensitive menu is used to perform the commitment task. Once selected, the **Commit GamePlan status to Gallery** dialog box appears as shown below.

The three main status indicators are defined as *GamePlan Created*, *VS Project Created*, and *Project Checked-In*. If needed, for whatever reason, the Migrator can select *Reset Internal Programs* to change the status of internal programs in the GamePlan back to "internal" in the Gallery thus removing the "lock" from the object icon.



A section is available to add migration notes that will be updated into the Gallery Notes tab for each GamePlan object. Notice the standardized placeholders provided to make adding comments easier. To insert a placeholder, simply press the Symbol button. In the example above, "GP {0} created {3} from {1} by {2}" results in "GP SDP100C created 9/21/2007 from MIT3116 by agardner" being added to the Notes tab of the GamePlan objects.

This page is intentionally left blank.

Chapter 15 - DataGate for iSeries and ADO: a Perspective

For scalable client/server access to the iSeries, ASNA's DataGate for iSeries[1] (DG/400) has been the most successful middleware product available. Hundreds, if not thousands, of Windows- and Web-based applications have been developed utilizing DataGate for iSeries. As further testament, these applications have been *deployed* to over a million end users.

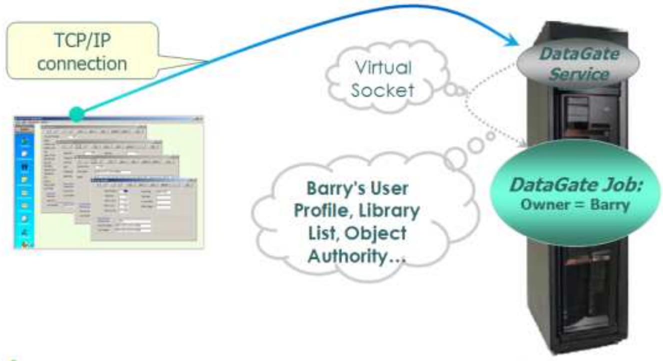
The success, even the elegance, of DG/400 is in its utilization of the native iSeries database engine, DB2 UDB[2]. By employing the same APIs as the iSeries' RPG, COBOL, and CL languages, DG/400 is able to capitalize on the strengths of DB2 UDB. Client/Server connectivity is via TCP/IP, which is ubiquitous and inherently supported by both client and server machines.

The ability to provide direct support for RPG I/O commands and built-in functions employed by business systems nearly guarantees the integrity of migrated applications.

How DG/400 Works

This overview of DataGate's functional characteristics illustrates the security and mechanics that provides client applications native, record-level access to the iSeries.

Establishing a Connection



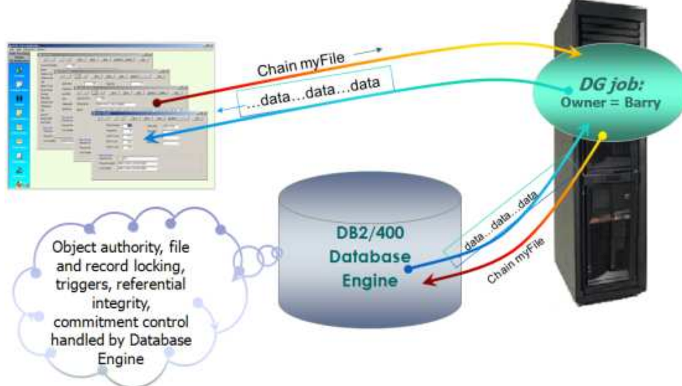
When a client application requests a connection to the iSeries[3], ASNA DataGate Client passes several IP packets to the DataGate Service running on the iSeries. Encrypted in these packets is the logon ID and password used by DataGate to confirm credentials and to establish the user's object authority and the application's library list. When the connection process is completed, an iSeries job exists to service the client application. All subsequent iSeries object access performed by the client will be under the authorities provided by the connected user's profile.

[1] Also known as DataGate/400

[2] Originally named "DB2/400"

[3] Visual RPG command: "Connect"

File I/O Operations



When the client application executes a Visual RPG I/O command (i.e., OPEN, SETLL, READ, READE, READP, UPDATE, WRITE, COMMIT, etc.), the operation request is sent to the DataGate job. The job then formulates the DB2/400 API Function Call to perform the task. The DB2/400 API returns the record to the DataGate job that sends the record to the client application. Incidentally, the DB2/400 API functions called by DG/400 to implement I/O commands are virtually the same API functions called by compiled RPG/400 and RPGLE programs.

The DataGate job maintains the file connection, and therefore appropriate record locks, until the client application closes the file. Valid indexed[4] and arrival file access are supported for physical, logical, and SQL view files.

CALL/Parm Operations

DG/400 provides OS/400 CALL mechanics to the client application. Any non-interactive iSeries program[5] can be called by the DataGate client, including system APIs. DG/400 supports single direction (for performance) in addition to bidirectional parameters of valid data types, including data structures.

Open Query

Visual RPG surfaces open file query via properties of the database file object[6]. DG/400 implements open query by setting the QRYSLT and KEYFLD parameters of the iSeries' OPQRYF command. DataGate for SQL Server (DSS) implements open query by setting the SELECT WHERE clause of a SQL command. In either case, the query executes when the file is opened, and the file path is automatically overridden to the resulting query file.

Translation

Data must be translated between the .NET client machine (Unicode) and the iSeries (EBCDIC), which is handled by IBM APIs provided in the OS/400 operating system.

Scalability

Each client connection is supported by a DataGate job dedicated to the application as long as the connection is maintained. Although DataGate jobs are assigned to the QINTER

---

[4] By Key or by RRN

[5] Type *PGM

[6] DclDiskFile: QryFileName, QryKeyFlds, QrySelect

subsystem by default[7], DataGate can run in any subsystem, including its own. Although DataGate jobs are submitted as batch jobs by default, they execute at priority level 20 in contemplation that the client applications are interactive rather than batch.

As ordinary iSeries jobs, DataGate jobs are mostly I/O and are not inherently CPU-bound. Functions that may contribute to CPU usage are programs called by the client application, EBCDIC-Unicode translation services, and DB2/400 APIs.

Network Performance

DataGate provides control of data volumes over the client/server network with features called *network blocking*, *cache writing,* and *directional parameters* (CALL/Parm).

Network blocking allows the developer to request DataGate to send blocks of data records to the client for files the application is processing sequentially. It is common knowledge these days that transmitting multiple records in a single I/O request-response transaction over the network is faster than transmitting the same records individually. The developer can specify a fixed number of records blocked at one time via the BlockingFactor property of a data file definition, or variably by employing Visual RPG's "range"[8] commands.

Caching records is an option that, when combined with network blocking, writes multiple records to an output data file in a single TCP/IP transaction rather than one record at a time. When implemented without network blocking, the CacheWrites file property can instruct the database engine to employ its native disk caching[9].

By definition with OS/400's CALL/Parm, parameters are passed by *reference*. This means that parameter values are marshaled to and from the CALLed program. Typically, many program parameters have context to the application in only a single direction. ASNA DataGate Client allows program calls to be executed where the data is passed to the server only, from the server only, or from and to the server. For instance, if a 4K character data structure parameter is passed only from the server, then DataGate can be instructed to omit the 4K in the program CALL to the server.

Client/Server vs. Host Based

Traditional, green-on-black iSeries applications have the advantage of being run on the same machine as the database engine and data store. This environment is so typical and commonplace that it frequently becomes problematic for iSeries developers even when they are developing completely new client/server applications in Visual RPG for .NET.

The foremost constraint in client/server processing is the network. Software engineers from all disciplines know that with client/server they are dealing with the physical aspects of packet transfer, task switching, bandwidth, traffic congestion, and so on. Client/server is well suited to transaction processing, especially when these features can be employed.

[7] Subsystem assignment is a installation option

[8] SetRange, ReadRange and DeleteRange

[9] Has no affect with DB2/400

As a resource, refer to Client/Server Software Architectures—An Overview published by Carnegie Mellon Software Engineering Institute[10]. Note the entirety of the article refers to *transactions*.

Batch and batch-like processing is problematic with client/server because of the high data volumes and the high number of I/O requests. Developers know that they must confine the logic for this type of processing to the database servers. The logic pieces that are data- and I/O-intensive are often implemented on the database server so that they may be called remotely from the client. In database servers, these logic components have been referred to as "stored procedures". With DataGate/400, the logic components on the iSeries are Callable programs. Isolating batch processing in the database server has a positive effect upon program performance and minimizes network traffic.

Record-Level Database Access vs. Set-based Database Access

The design of the DB2 UDB database engine supports record-level access so well, that both RPG and COBOL languages for the iSeries have always provided an array of record-level Input/Output commands and functions. Although embedded SQL support was added to the RPG and COBOL languages, its usage has been very limited relative to the amount of code that has been written in the years since its introduction.

Which is the Best One to Use?

On the iSeries, neither record-level nor set-based data access is a clear winner over the other. Each has its relative strengths and weaknesses[11].

SQL has had the advantage when:

1. Selecting and manipulating groups of data. For instance, changing or deleting a group of records in a single statement.
2. Tallying and calculating columns of data. Averaging and math functions, e.g., can be performed during the record selection phase.
3. Aggregating columns. For instance, to find a list of different postal codes and count how many addresses were in each postcode.

Native I/O has had the advantage when:

1. Accessing many discreet or unrelated files, as in *transaction* processing.
2. Updating, adding, and deleting records asynchronously rather than in groups.
3. Maintaining pessimistic record locks.

[10] http://www.sei.cmu.edu/str/descriptions/clientserver_body.html

[11] Source: Embedding SQL in RPG IV by Joel Cochran

Which Performs Better?

Until recently, the iSeries SQL engine has been a poor performer for tasks more complex than the examples noted above. Another area where the SQL engine has performed poorly (also meaning excessive consumption of CPU resources) has been when serving multiple users. Whether host-based or client/server via ADO, SQL performance could be tolerable for one or two users, but for more than just a few users, performance has been intolerable.

The historic inability of the iSeries' SQL engine to scale for multiple users is illustrated in the ADO and DG/400 comparison benchmark document, Fast Access to DB2/400[12]. Single client tests (e.g., Test1a) demonstrated very good performance from the SQL engine when processing read-only record sets. However, when updating or adding records, the SQL engine labored. Finally, with file updating and adding (i.e., "transaction" processing) the SQL engine performed very poorly and ran CPU usage to the maximum.

Clearly, native I/O continues to perform very well on the iSeries. In addition, this explains the success of DataGate for the iSeries: DG/400 drives the DB2 UDB database engine just as regular RPG programs do.

Client/Server Differences

An important client/server distinction between IBM's ADO Provider and ASNA DataGate Client is:

> **SQL engine** - the logic incorporated into the SQL command is executed on the server machine. The resulting record set or scalar is transmitted to the client.
>
> **DataGate/400** – as with all native I/O, the logic is executed in the application code. Typical transaction processing remains very fast (within the limits of network bandwidths and traffic). However, when processing extreme volumes of data (i.e., "batch" processing), it is possible to extract better performance via SQL.

The iSeries SQL Engine Improves

The bright news for SQL developers is that beginning with OS/400 V5R3 and most remarkably, with V5R4, IBM has made significant improvements to the iSeries' SQL engine. Not only is performance better, but tolerance for multiple, concurrent users has improved dramatically. From V5R4 forward, enterprise developers are now more able to employ embedded SQL without as much fear of adversely affecting the system's performance.

It is important to note that it is neither practical nor necessary to use one data access mechanism to the exclusion of the other. Each has its own strengths as noted above in Which Performs Better? From a client/server implementation, when SQL makes more sense, then use ADO.NET with IBM's .NET Provider. On the other hand, when Native I/O makes more sense, then use ASNA's DataGate for iSeries. Finally, it is important to note that *both* access mechanisms can be employed side-by-side in any .NET class.

---

[12] Tests conducted with OS/400 V5R3

DataGate's Role in Monarch

Monarch's strength lies in its ability to maintain more of the application's original logic than any other migration product for the iSeries. Recognizing the meat of a business application is in its data handling logic, DataGate/400 is Monarch's greatest asset in that it allows the data handling to be migrated without alteration. Monarch enables the migration of thousands of program objects in a very short time because the business logic does not change.

If RPG's I/O commands were to be re-factored into SQL commands, the migration process would be long and protracted, and it would require an extraordinary amount of manual re-coding to interpret the program logic. Imagine a SETLL at line 3285 that precedes the execution of a READE on line 1039! Because Monarch migrates RPG to RPG and DG/400 retains the original native I/O, the juxtaposition of SETLL and READE is not a concern to the Migrator.

Consider that the legacy application was built with, and ran on, a host-based, green-on-black environment with RPG and CL program objects executed by the OS/400 operating system. The newly migrated application is built within Microsoft's .NET Framework running as browser-based served by a Windows 2003 Server that is accessing data from the iSeries or SQL Server, or both.

It can be reasonably anticipated there will be "hot spots" where pieces of the legacy application may not fit well into the new environment. Although the migrated programs execute as they had before, the UI appearance of some or response time of others may need to receive individual attention of the migration team. For instance, you might expect to replace some of the application's subfiles with Grid web controls, or date fields with calendar controls, and so on.

Similarly, it is expected that some batch-like programs that performed tolerably in a host-based environment may not execute well in a client/server environment. The rule of thumb is that batch logic should remain on the data server. For the iSeries, this can be implemented in a client/server environment either by CALLing the iSeries-resident program or by employing ADO .NET.

Conclusion

Without DataGate for iSeries, the migration of the thousands of RPG programs that comprise a business system would be length, labor intensive, and costly. Re-factoring the I/O logic in *every* one of the programs can easily introduce logic errors in otherwise well-tested code.

With DataGate for iSeries, the migration of the I/O logic remains intact. Remediation of the few programs where processing is indeed more batch than transaction can be undertaken in a more timely and safer manner.