



**ASNA MONARCH<sup>®</sup>**

---

## **User's Guide to Migrating Applications**

---

An Introduction to the Concepts and Process  
of Migrating Applications using ASNA  
Monarch.



---

**ASNA**

**ASNA Inc. - USA**  
9901 West IH-10, Suite 1000  
San Antonio, Texas 78230  
USA

**800-289-ASNA (2762)**  
**210-408-0212**  
**210-408-0211 Fax**

**ASNA Ltd. - Europe**  
31 Frederick Sanger Road  
Surrey Research Park  
Guildford, Surrey GU2 7EP  
England

**+44 1483 570666**  
**+44 1483 578191 Fax**



## Table of Contents

<b>Chapter 1 – Introduction</b> .....	<b>1</b>
1. Object Inspection .....	2
2. Visual Analysis and Strategy Generation.....	3
3. Source and Data Migration.....	4
4. Unsupported Features Resolution .....	5
Epilogue.....	5
<b>Chapter 2 – Preparing Galleries</b> .....	<b>7</b>
Object Gallery.....	7
Creating a Gallery .....	7
Dealing with the Source .....	10
Source Search Algorithm.....	11
Checking the Source Mappings.....	12
Discovering Copybooks .....	14
Discovering Printer O-Specs .....	16
<b>Chapter 3 – Working with GamePlans</b> .....	<b>20</b>
Partitioning an Application .....	20
Project Types.....	21
Interactive .....	22
ASP .Net Web Application .....	22
ASP .Net Class Library .....	23
Non-Interactive.....	24
Console Application .....	24
Class Library.....	25
Finding Entry Points.....	27
Creating GamePlans .....	28
Including Additional Objects .....	30
Establish GamePlan Directives .....	31
Origin .....	32
Visual Studio Options.....	32
External Field Reference Description .....	33
Display File Function Keys .....	33
Data File .....	33
New .NET Print files .....	34
Checking the Task List .....	34
Problems.....	35
Unsupported Features.....	35
Exceptions.....	36
Sizing a Migration.....	36
Magnitude Density.....	36

Algorithm .....	36
Customizations .....	38
Effort Density .....	39
Algorithm .....	39
Customizations .....	41
<b>Chapter 4 – Pulling the (Migration) Trigger .....</b>	<b>43</b>
Establishing Templates .....	43
Solution Builder .....	44
<b>Chapter 5 – Migrating Display Files.....</b>	<b>45</b>
Templates .....	45
Template.aspx .....	46
Template.aspx.vr .....	46
Example.....	46
<b>Chapter 6 – Migrating Printer Files .....</b>	<b>49</b>
Interesting Keyword translations.....	50
<b>Chapter 7 – Migrating CL Programs .....</b>	<b>51</b>
Resources .....	52
<b>Chapter 8 – Migrating RPG Programs.....</b>	<b>53</b>
Setting up Migration Preferences .....	53
Import Options .....	53
Ignore Comment .....	53
Ignore Slash '/' Lines .....	54
Include Banner .....	54
Include Copybooks .....	54
Ignore Ln Comments .....	54
Include Messages .....	54
Use Symbol Operators .....	54
Comment Pad .....	55
Styling Options .....	55
Stylize White Space.....	55
Create Definitions.....	56
Relocate Global.....	56
Convert to Assignment.....	56
Use Symbol ANDOR.....	56
Indent Character.....	56
Preferred Subroutine Call Format .....	56
Op Code and Keyword Case.....	56
Program Migration Directives .....	57
Origin .....	57
<b>Gallery</b> .....	<b>57</b>

Shows the fully qualified name of the gallery used to create the GamePlan where this program exists .....	57
Internal .....	58
Remote.....	58
External .....	58
Op-Code Support .....	59
Normalization of RPG Op-codes.....	59
RPG Compiler Directives.....	60
Loading Program References .....	61
Generation of Migrated Code - Behind the Scenes .....	61
/Error AVR compiler directive.....	61
Migrated Visual RPG File's Header .....	62
New RPG Program's Wrapper Class.....	62
ASNA.Monarch.Program Base Class .....	64
ASNA.Monarch.Job class .....	66
Monarch-Generated Properties.....	66
Local Data Area.....	68
iSeries Data Area access .....	69
Status Data Structures .....	70
Program Status.....	72
File Status.....	72
Array Translation .....	72
Hexadecimal Constants.....	73
Compile-Time Data.....	73
AVR Program Constructor.....	73
Generated compile-time loading subroutines .....	74
Markers Supported as Start of "compile-time data" .....	74
Other Considerations .....	75
Cycle .....	75
Embedded SQL .....	75
Advanced RPG ILE considerations.....	75
Internally-Described Files .....	75
<b>Chapter 9 – Advanced RPG Data Structure Migration .....</b>	<b>76</b>
Overlapping Data Structures.....	76
Data Structures with Gaps.....	76
Giving individual field names to elements of an array .....	76
Masquerading a Data Structure as a Large Character Field .....	78
Partitioning a Data Structure fields into sub-fields (Date, Time) ..	79
Partitioning a Data Structure Fields into Sub-Fields.....	80
Renaming fields and mapping to arrays.....	81
Externally-Described DS augmented by new fields.....	83
Unsupported Overlapping Data Structures.....	84

Other Code Generated by RPG Agent .....	84
Printer O-Spec overflow and conditional EXCEPT support.....	84
Print related code (Net Print File defaults) .....	85
Insert “/Error Please set printer name” in Open Statements ..	85
Use this printer name in migrated .Net Print files: XXXX.....	86
Generate code to set the “Printer” property in migrated program, using: YY.....	86
Don’t send output to printer, instead leave manuscript in pathname: PPP .....	86
<b>Chapter 10 – Migrating Printer O-Specs.....</b>	<b>88</b>
Discovering Printer O-Specs .....	88
Monarch-Generated Record and Field Names .....	92
Record Formats without a Name .....	92
Conditioned Field lines .....	92
Migration Directives.....	94
Margins Group .....	96
Font Group.....	96
Paper Kind Group .....	96
Orientation Group .....	97
Overflow .....	97
Restore Defaults Button .....	97
RPG Agent Handling of Printer O-Specs.....	97
Substitution of Except for Write.....	97
Handling Page Overflow .....	98
The Overflow Indicator.....	98
Fetching Overflow versus Being Conditioned on Overflow .....	98
<b>Chapter 11 – Migrating Message Files.....</b>	<b>102</b>
Install Directory .....	103
<b>Chapter 12 – Migrating Data.....</b>	<b>104</b>
Using Database Manager.....	104
Copying a Complete Library.....	104
Copying a Single Physical File .....	105
Copying a Single Logical File.....	105
Considerations for MS SQL Server .....	106
<b>Chapter 13 – Resolving Unsupported Features .....</b>	<b>108</b>
How to Migrate GOTO from a Subroutine to Mainline Code .....	108
How to Migrate Embedded SQL .....	110
<b>Chapter 14 – Deploying the New Application .....</b>	<b>113</b>
Installing Application Message files.....	113
Monarch application deployment is no different from any AVR Web application .....	114

---

Folder Structure.....	114
Additional server components needed.....	115
Deploying your project - A little more detail.....	116
Using Visual Studio's Web Setup Project to Deploy an AVR for .NET Web application .....	117
Creating an MSI Installation File.....	117
Special considerations when deploying on Windows 2003 Server....	118
Printers Available for Monarch Applications.....	119
<b>Glossary .....</b>	<b>120</b>

This Page Intentionally Left Blank

## Chapter 1 – Introduction

After the decision has been made that it is necessary to migrate away from the 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:

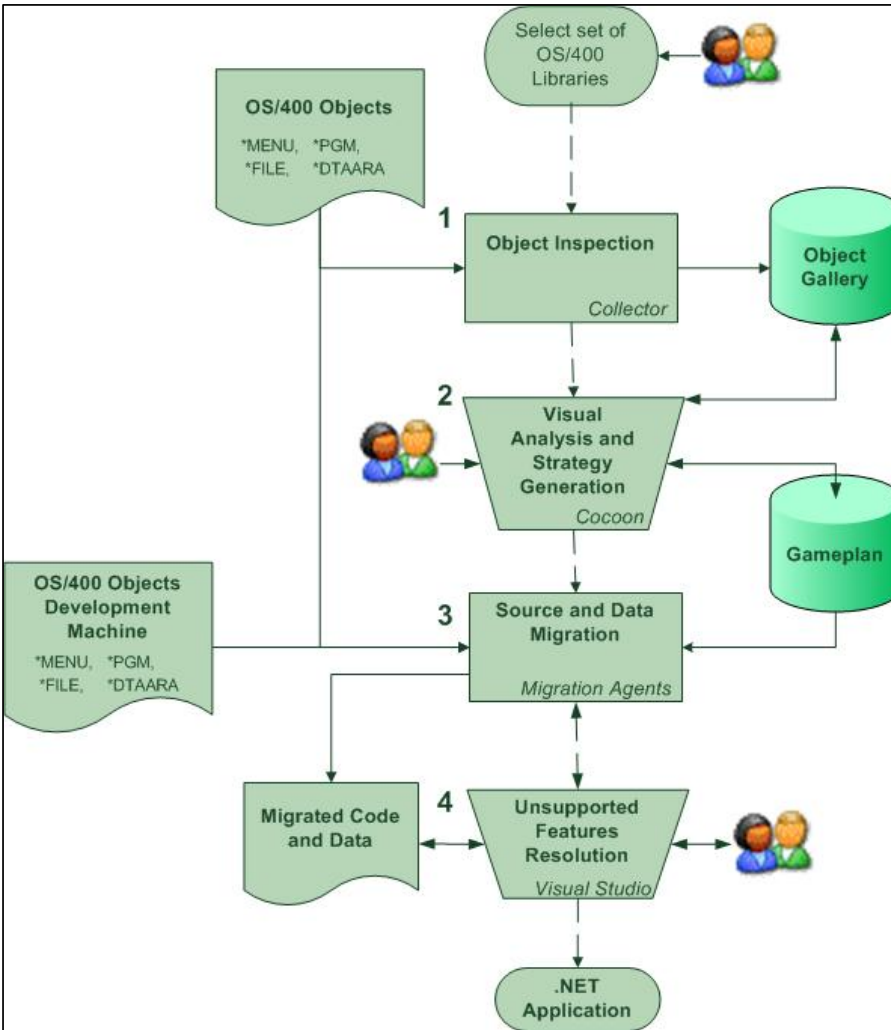
- Identification of Applications to Migrate.
- High-level description of the Applications.
- Team roles and responsibilities.

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, which will be discussed in this chapter.

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

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
- 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 these step later on. However, it is possible to add more libraries after the initial collection has occurred.

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.

See also [Chapter 2 – Preparing Galleries](#).

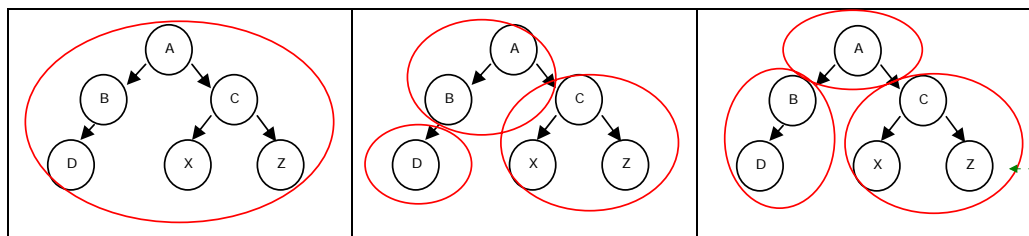
## 2. Visual Analysis and Strategy Generation

Important decisions must be made to determine how 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.

The Object Gallery may be broken down into several **GamePlans** to facilitate object migration and re-use (redundancy avoidance). **Each migrated GamePlan constitutes a Visual Studio 2003 Visual RPG Project.**

A GamePlan is defined by one or more entry points. Each entry point is a Program that can be called from other Programs, Menus or the command line. The GamePlan includes not only the Programs called by the entry point Program(s), but also all other objects used by the Programs involved. Here is an example of how a six Program application can be divided into GamePlans in several different forms.



**Formatted:** Centered, Space  
Before: 0 pt, After: 0 pt

Each GamePlan has associated with it a set of properties, including a Project Type, a Name and a Location. These properties are used to create a Visual Studio **Project** where the Subsystem will be 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 into GamePlans.

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

Cocoon allows both the Object Gallery and the GamePlan to be opened and made visible at the same time. Drag-and-drop operations to organize objects are available to the Migrator. The state of the Visual Analysis session persists in the form of a Migration **Solution**.

To create a GamePlan, an entry-point or top-level Program is selected from a Gallery and a Library List is supplied. The Cocoon will crawl thru the object reference web to determine all objects forming the Subsystem. This information is the basis for the initial content of the GamePlan. If a GamePlan turns out to be too big or cumbersome to manage as a single entity, it is possible to partition it by pruning a branch from the call graph and creating a new, smaller GamePlan out of the branch. Cocoon will then place a reference where the branch used to be pointing to the new GamePlan.

See [Chapter 3 – Working with GamePlans](#).

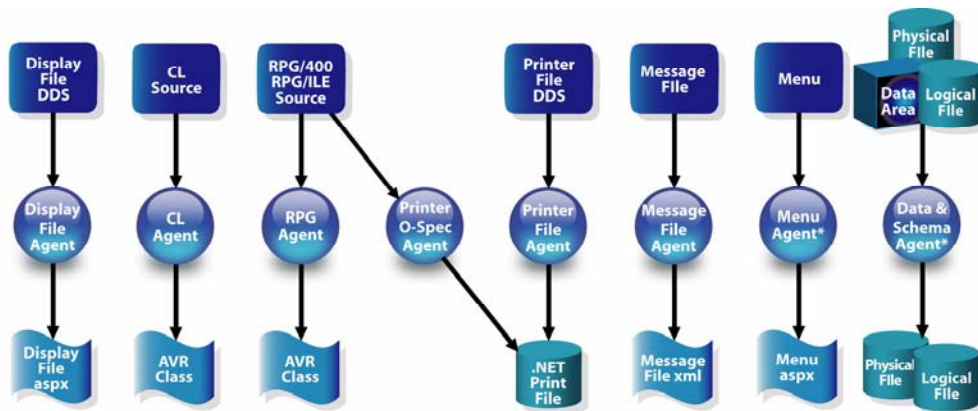
### 3. Source and Data Migration

Once the migration Directives are defined in the previous step, the actual migration of the Application is performed by selecting Cocoon's **GamePlan – 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 and Errors will be separately logged to assist during the conflict resolution step.

There are multiple agents used to Migrate specific object types and one agent responsible of 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 Cycle or of ICF files. Also, eliminate the code that is irrelevant in the new environment.
- Study any compiler 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 have to be incorporated in 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.

This page intentionally left blank.

## Chapter 2 – Preparing Galleries

### Object Gallery

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 thru a set of libraries on the OS/400 system to collect data for each object (e.g.: Program, File, Data Area, Menu) 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**.

*Hint: Using a DataGate Windows database (local or multi-user) in lieu of an iSeries has these advantages:*

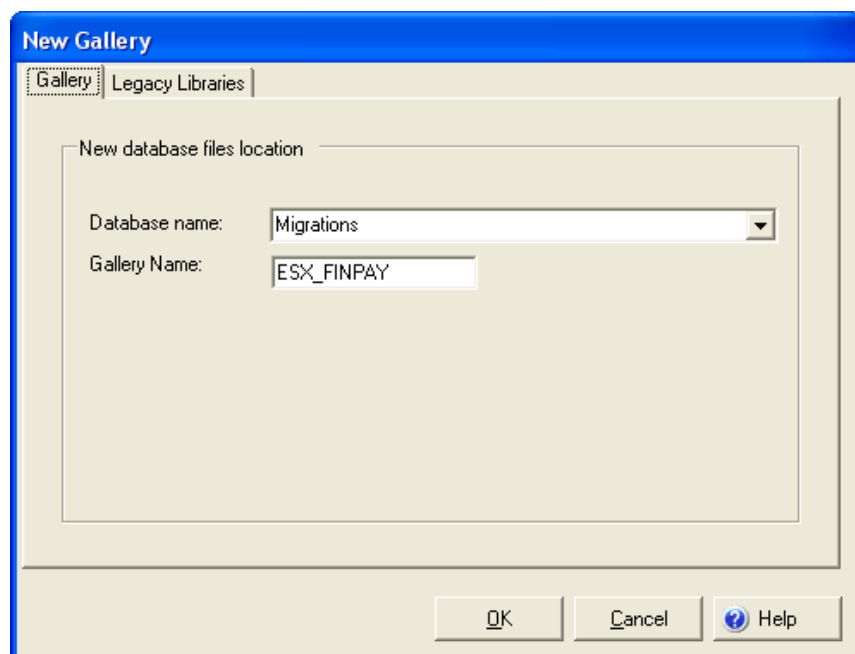
- No additional libraries on the iSeries.
- Gallery names may be 31 characters in length rather than 10.

### Creating a Gallery

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

**To Create a Gallery**, select the **File → New → Gallery** menu option from Cocoon. 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 Server (which is running DataGate). The name of the Gallery should be representative of what it will contain. If you plan on storing the Gallery on an iSeries database, then select a name no longer than 10 characters. **Remember, a Gallery becomes a Library in the selected database.**

The following figure shows the dialog used to create a Gallery called **ESX\_FINPAY** on a database called **Migrations**.



Notice that the database name can be selected by clicking on the drop-down box, which is populated with all the database names registered on your machine.

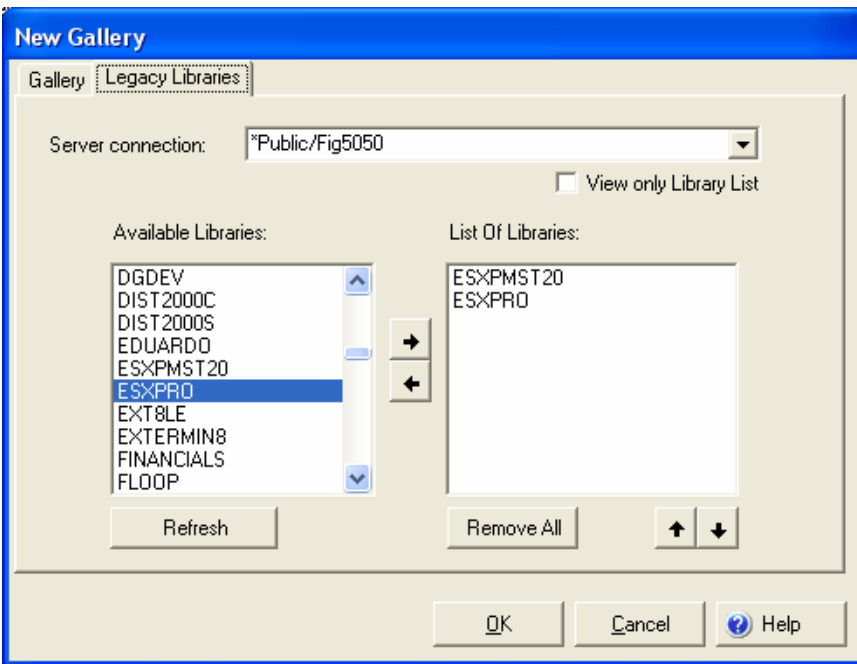
*Hint:* To create new database names you can use *DataGate Database Manager*.

In addition to stating the name and location of the Gallery, you have to 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 an iSeries.

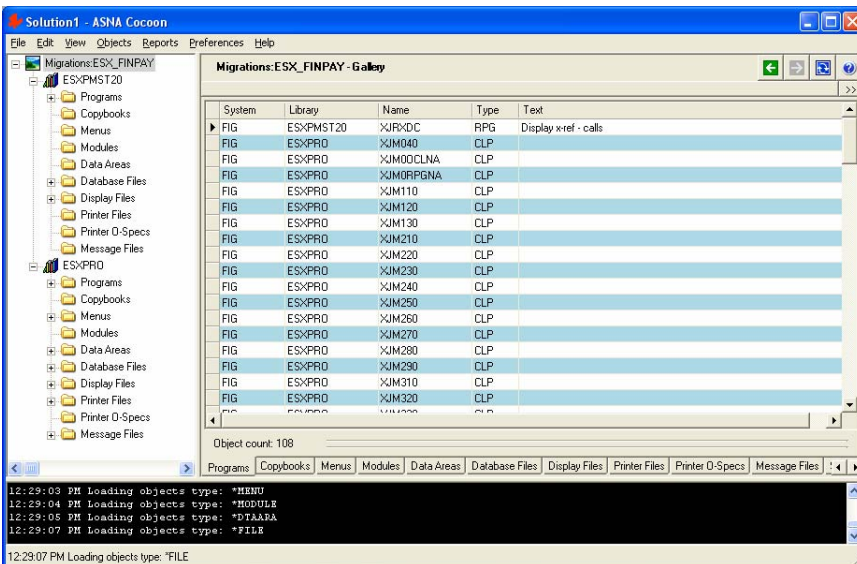
The second tab in the New Gallery dialog is labeled **Legacy Libraries**. In this tab, you select a connection to an iSeries via a DataGate database name and a list of libraries from that system. It is necessary to have the ASNA Monarch Collector Program installed on the selected server.

Once you are satisfied with the data entered, and the OK button is clicked, ASNA Cocoon creates the Gallery by invoking the Collector on the iSeries 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. Periodic reports will show the progress being made while the collection process is running.

The following figure shows the libraries **ESXPMST20** and **ESXPRO** being selected from the **Fig5050** database connection.



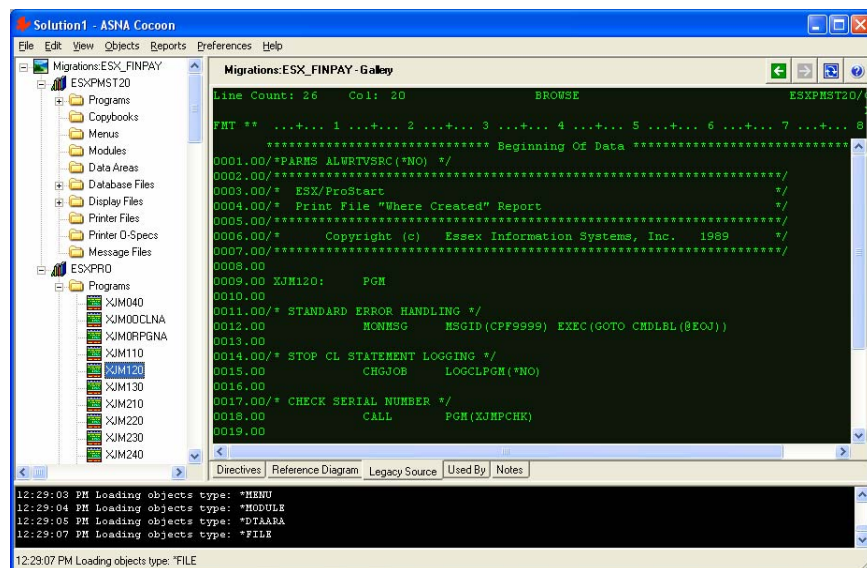
After the collection process finishes running, Cocoon presents you with a list of the objects found in the requested libraries. The objects are grouped by type within each Library. By clicking on the Solution Explorer (the left pane shown in the figure below) you can subset the objects shown in the right pane to only those in a particular Library or an individual object.



V 0.909

If you have selected the whole Gallery or a particular Library within the Gallery in the Solution Explorer, you can click on the Tabs at the bottom of the right pane, which allows you to see different object types.

When a particular object is selected from the Solution Explorer, such as Programs, a set of tabs in the right pane allows you to get information on the selected object. The set of tabs shown is dependent upon the object type. However, there are three tabs that are always present; **Directives**, **Used By** and **Notes**. Many object types also show a **Legacy Source** tab as shown in the next figure:



**Note:** Once a Gallery is created, you can add more Libraries from the same server by selecting the **Object → Add libraries to Gallery** menu option.

## Dealing with the Source

It is not uncommon for the source to be moved to different libraries or even different files during the lifespan of an application. The mechanism Cocoon employs for locating the source is based upon a match with the Original location, which results in the target Database, Library, File and Member where the source now resides.

Finding the source for a particular object can be tricky. When an object is created from source, like Programs and file, 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.

When Cocoon needs to get to the source, the **Registered** source information is used to locate it. The basic information known is:

1. System Name.
2. Library Name.
3. File Name.

#### 4. Member Name.

This information is known as the **Original Source Location**. 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 name pointing to the OS/400 System. This database name should use a User Profile with enough authority to read the source's member records.

Cocoon keeps a **Source Mapping** table where an association is made between the original source locations and the database name to use to get to the source. ***This table is important because names change through the development process of an application, it is possible that the original Library, file and member names are not current any more.*** The Source Mapping table allows you to inform Cocoon of the **new** names where the sources can be found for particular objects or groups of objects.

Entries in the Source Mapping table can be very generic, as when only a system name is mapped to a database name. The wild card '**\*ALL**' is used to serve as place-holder when entering generic entries in the table. Using \*ALL 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 available for each **Gallery** and **GamePlan**.

	Original Sys Name	Original Library	Original Filename	Original Member	Database Name	Library	Name	Member
▶	S104C54A	EXSPDEV	EXPDDSSRC	RDS035DF	CHY_5050	EXPMST20	QDDSSRC	RDS035DFM
	S104C54A	EXSPDEV	EXPDDSSRC	*ALL	CHY_5050	EXPMST20	QDDSSRC	*ALL
	*ALL	WH32MAT	QRPGLESRC	*ALL	CHY_5050	WH32MAT	QRPGSRC	*ALL
*	*ALL	*ALL	*ALL	*ALL	CHY_5050	WH32MAT	*ALL	*ALL

Object count: 4

Directives | Call Graph | Effort | Magnitude | Task List | Object Usage | Notes | Source Mappings

### Source Search Algorithm

The figure above will help illustrate how mapping works:

1. The **Original Member** RDS035DF of the **File** EXPDDSSRC in the **Library** EXSPDEV that resided on the iSeries named S104C54A is now found as the source **Member** RDS035DFM of file **Name** QDDSSRC in the **Library** EXPMST20 that resides on the iSeries name that is accessed by Monarch with the **Database Name** of CHY\_5050.
2. All source Members of the File EXPDDSSRC in the Library EXSPDEV that resided on the iSeries named S104C54A is now found as the *same* source Member of file QDDSSRC in the Library EXPMST20 that resides on the iSeries name that is accessed by Monarch with the Database Name of CHY\_5050.
3. All source Members of the file QRPGLESRC in the Library WH32MAT that resided on any iSeries is now found as the *same* source member of file

ORPGSRC in the Library WH32MAT that resides on the iSeries name that is accessed by Monarch with the database name of CHY\_5050.

4. All source Members of any file name in any Library that originally resided on any iSeries is now found as the *same* source member of the *same* file in the Library WH32MAT that resides on the iSeries name that is accessed by Monarch with the database name of CHY\_5050.

The position of the mapping entry 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 **exact** original name:

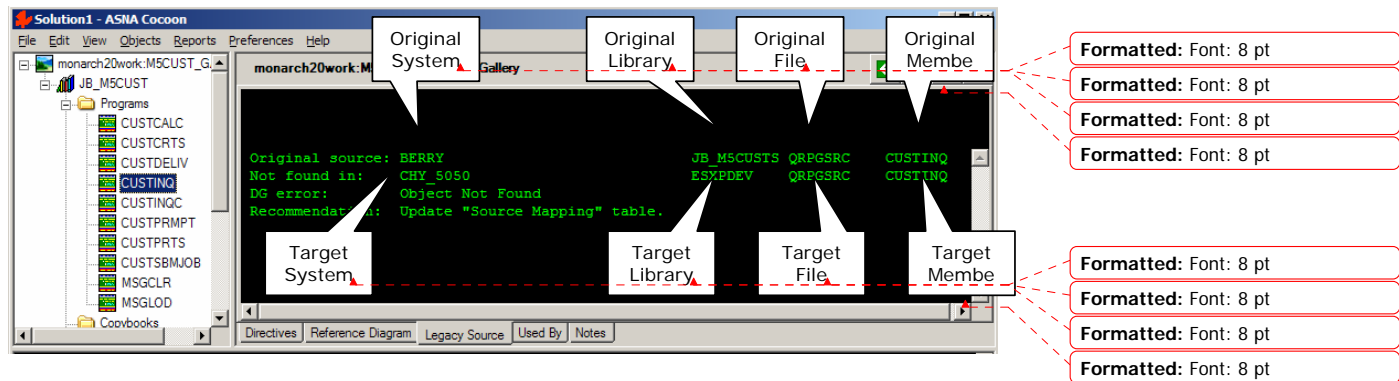
Original System	Original Library	Original File Name	Original Member
Match	Match	Match	Match
Match	Match	Match	Any
Match	Match	Any	Match
Match	Match	Any	Any
Match	Any	Match	Match
Match	Any	Match	Any
Match	Any	Any	Match
Match	Any	Any	Any
Any	Match	Match	Match
Any	Match	Match	Any
Any	Match	Any	Match
Any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

### Checking the Source Mappings

An easy way of finding out if the Source Mappings table is set up correctly is to ask Cocoon to report any source file it **cannot** find. This is done by selecting the **Object** → **Discover Object Source** menu option while having the Gallery selected from the

Solution Explorer. After you have run this option, the name of each object for which the source could not be found is displayed in **red**.

If you get a red object, click its **Legacy Source** tab and you will find the Original Source Location and the current location used. You can correct the Source Mapping table by editing an existing entry, or by adding a new one for the offending object.



*Hint: It is advisable to set up the Source Mappings once for the Gallery, as its values are inherited when you create a GamePlan, thus preventing you to fix them for each individual GamePlan.*

## Discovering Copybooks

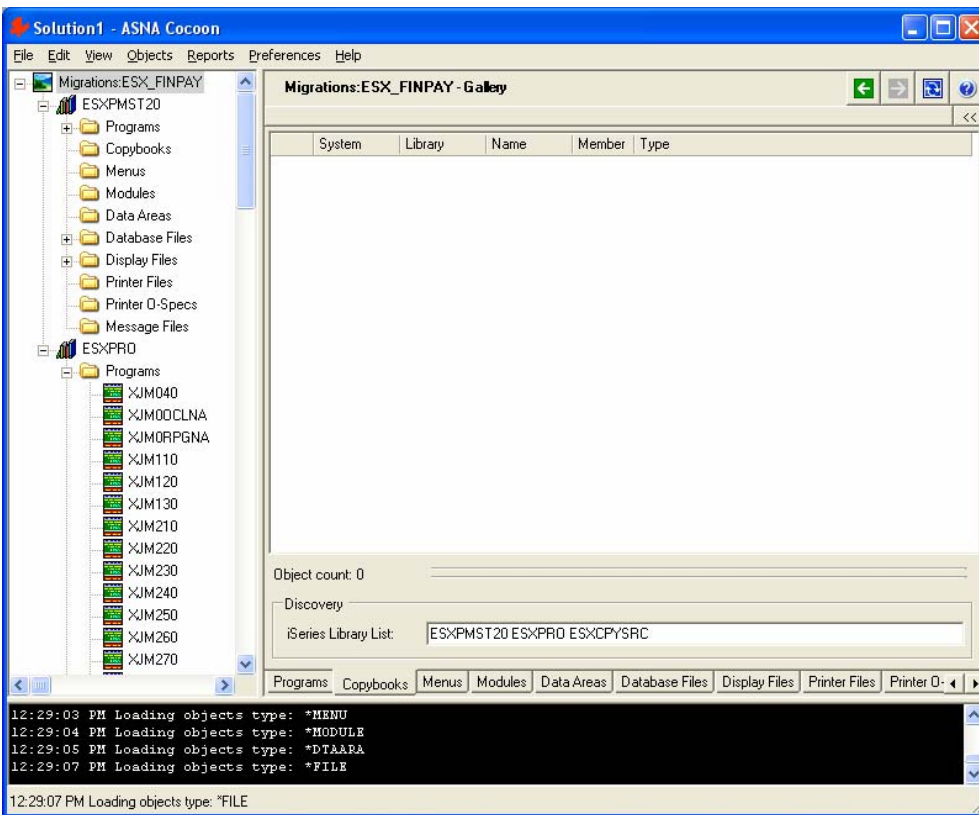
Once the Source Mappings are correct, the next step needed to set up a Gallery is to discover the Copybooks used by the RPG Programs. This process, initiated by selecting the **Object** → **Discover Copybooks**, reads all the sources of the RPG Programs searching for the **/COPY** Directives.

Even though Copybooks are not strictly OS/400 objects (but are separate chunks of code used in one or more Programs), Cocoon treats them as if they were objects to facilitate the task of setting Directives on how to treat individual Copybooks.

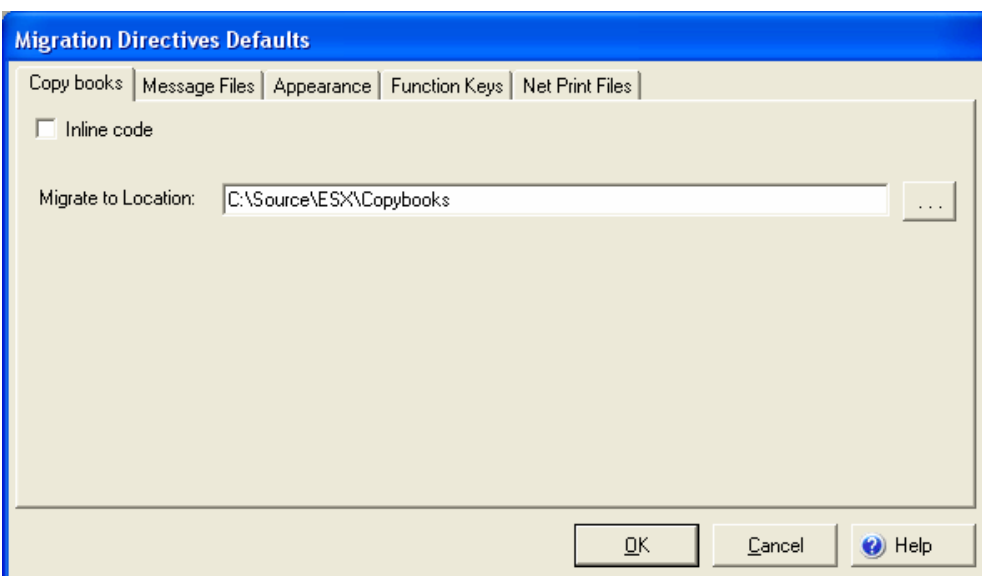
The **/COPY** on OS/400 allows a fully-qualified member to be specified by providing the Library, File and Member names. It is also possible to omit the Library and/or File names. When the File name is omitted, QRPGRSRC is used for RPG/400 and QRPGLSRC for RPG ILE. The discovery process follows the same 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 iSeries. This Library list defaults to the libraries used when the Gallery was created, but it might not be in the proper order, or there might be additional libraries that must be included containing sources used by **/COPY**.

You can edit the Library list used by Discover Copybooks by clicking the **Copybooks** tab while selecting the Gallery in the Solution Explorer as shown in the following figure.



Once you have set up the Library list, select the **Object → Discover Copybooks** menu option in Cocoon. When you do, Cocoon will prompt you for the **default location** where you would like the Copybook sources to be located when migrating a GamePlan. This default location will be used as the initial **Directive** for each Copybook found in the discovery process. The following figure shows the location to be in the C drive under the **\Source\ESX\Copybooks** folder.



You can set the Copybooks Directives and specify if you want Cocoon to *inline* the Copybooks into the source. Selecting the **Inline code** option embeds the Copybook source into the Program by effectively removing the /COPY Directive, or to leave the /COPY Directive in place.

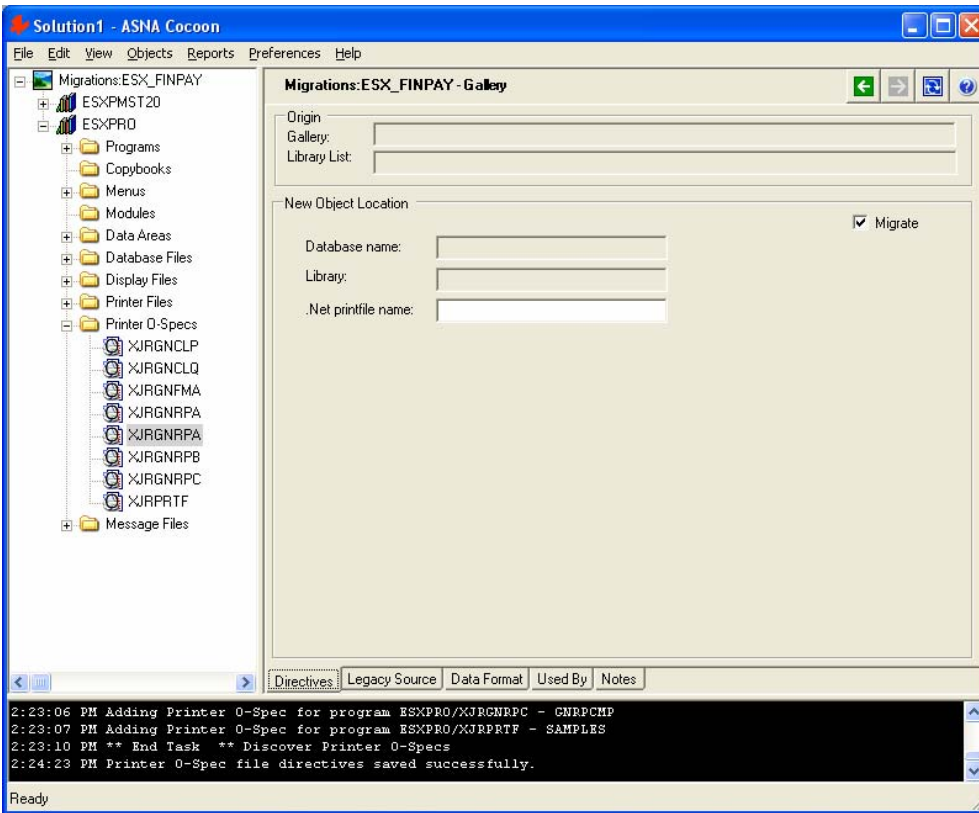
## Discovering Printer O-Specs

ASNA Visual RPG supports printing using externally-defined files. It does not 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. This process called Printer O-Spec discovery is initiated by selecting the **Object → Discover Printer O-Specs** menu option.

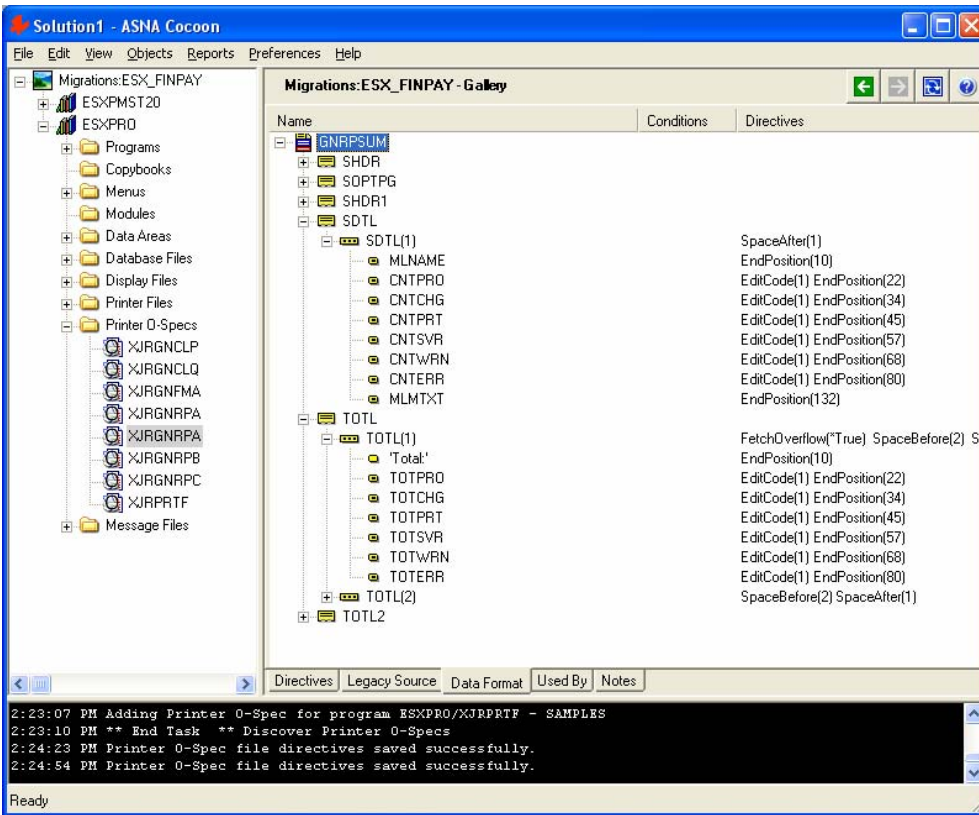
*Hint: It is important to have set up the Source Mappings before executing the process.*

Similar to Copybooks, Printer O-Specs are not OS/400 objects, but chunks of source. However, like Copybooks, Cocoon likes to deal with them as if they were objects in their own right, because after the migration occurs, they will indeed be objects. Each set of O-Specs will be migrated into a **DataGate** Printer File. Having them appear like objects facilitates the setting of Directives like names and locations.

The following figure shows the **Directives** page for a Printer O-Spec.



Notice the **Data Format** tab available for Printer O-Specs. It is used to show (in a formatted way) the elements that form the future Printer File. See the figure below.



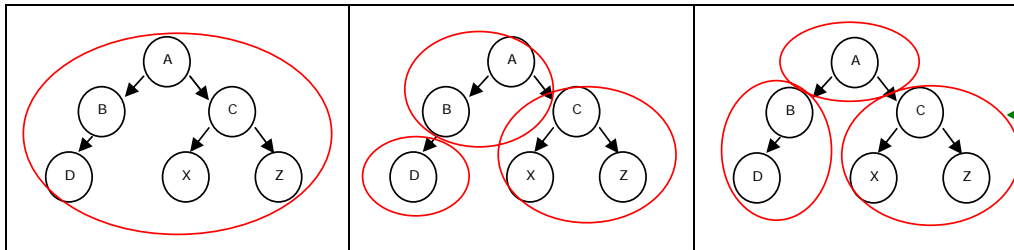
This page intentionally left blank.

## Chapter 3 – Working with GamePlans

### Partitioning an Application

An OS/400 application is composed of several Programs which call each other, forming a call graph that can usually be represented with a call tree. Conceptually, any part of the tree can be grouped into **Subsystems** of the Solution. There is a Subsystem that starts at the root of the tree and 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.

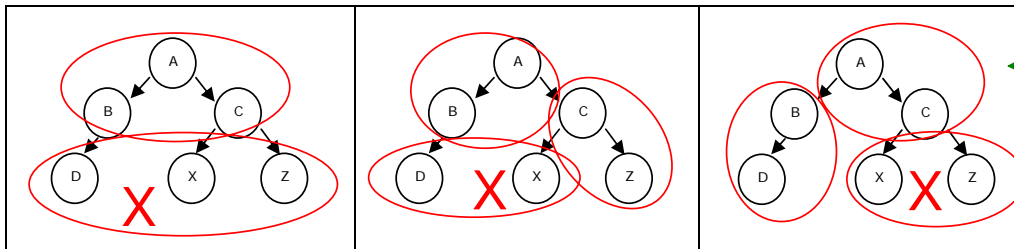
The following diagram shows three valid arbitrary partitionings of an application consisting of 6 Programs (A, B, C, D, X and Z). The smaller black round circles represent the Programs and the larger red ovals represent the Subsystems.



Formatted: Centered, Space  
Before: 0 pt, After: 0 pt

Formatted: Centered, Space  
Before: 0 pt, After: 0 pt

The following diagram shows three more ways of partitioning the Solution in which the Subsystems marked with an **X** have multiple entry points. These utility Subsystems are typically **Service Programs** on the OS/400 or **DLLs** in .NET.



Formatted: Centered, Space  
Before: 0 pt, After: 0 pt

When migrating an application, the Migrator will have to determine the partitioning into Subsystems and each Subsystem will become a migration Project organized as a GamePlan.

## Project Types

Monarch can migrate GamePlans into 6 different kinds of Projects. They are divided into **interactive** and **non-interactive**, depending if the GamePlan contains any Display Files.

The Migrator chooses the Project type according to how the **resulting assembly** is to be used. The following table defines the characteristics that should be considered in selecting the Project Type.

Project Type	Started by	Display Files	Entry points	Caller
<b>ASP.NET Web Application</b>	Browser Request	Yes	1	User
<b>ASP.NET Class Library</b>	Call/Parm	Yes	1 or More	Other Programs
<b>Class Library</b>	Call/Parm	No	1 or More	Other Programs
<b>Console Application</b>	Command Line or OSEXEC	No	1	User or Scheduler or Other Programs
<b>Web Service<sup>1</sup></b>	Web service Request	No	0	Web service client
<b>Windows (NT) Service<sup>1</sup></b>	Windows Services Console	No	1 or More	Service clients

The Migrator can create a GamePlan with a Project type of **None** which can be used to group OS/400 objects that get migrated once and which do not generate any kind of code. A GamePlan which 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.

---

<sup>1</sup> Available in Version 3.0

## Interactive

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

### ASP .Net Web Application

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 doesn't 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/Cocoon/Templates/AspWebApp

**Global.asax.vr**

```

BegSr Session_Start Access(*Protected)
  DclSrParm Sender Type(*Object)
  DclSrParm e Type(System.EventArgs)
  DclFld Device Type(Object)
  DclFld Job Type(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 = *New MyJob()
  Session("Job") = Job
  Device = Job.Start(*this.Session.SessionID)
  Session("Device") = Device

EndSr

```

**MyJob.vr**

```

BegClass MyJob Extends(ASNA.Monarch.WebJob) Access(*Public)
  DclDB Name(MyDatabase) DBName("**Public/Fig5047") Access(*Public)
  . . .
  . . .
  BegSr ExecuteStartupProgram Access(*Public)
    Connect MyDatabase
    Call --- Entry Point ---
  EndSr
EndClass

```

**ASP .Net Class Library**

It is a collection of Programs, some of which use Display files. One or more of these Programs are considered entry points and are called by other Programs outside of the class Library. The entry points into the class Library typically will 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	Yes

Caller	Other Programs
Started	Call/Parm

## Implementation

<b>Visual Studio Project Type:</b>	<b>ASP.NET Web Application</b>
Monarch Job for compile time:	A MyJob class is generated based on ASNA.Monarch.WebJob
Monarch Job for run time:	Uses the instance of the entry point ASP.NET web application (the caller)
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/Cocoon/Templates/AspWebDLL

### Global.asax.vr

```

BegSr Session_Start Access(*Protected)
  DclSrParm Sender Type(*Object)
  DclSrParm e Type(System.EventArgs)
  Response.Redirect("!NotCallable.htm", *True)
  LeaveSr
EndSr

```

### MyJob.vr

```

BegClass MyJob Extends(ASNA.Monarch.WebJob) Access(*Public)
  DclDB Name(MyDatabase) DBName("**Public/cherry")Access(*Public)
EndClass

```

## Non-Interactive

### Console Application

A console application is a 'Batch' 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 doesn't 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**.

### MyJob.vr

```

BegClass MyJob Extends(ASNA.Monarch.Job) Access(*Public)
  DclDB Name(MyDatabase) DBName("**Public/Fig5047") Access(*Public)
  BegSr ExecuteStartupProgram Access(*Public)
    Connect      MyDatabase
    Call         --- Entry Point ---
  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 use Display files. One or more of these Programs are considered entry points and are called by other Programs outside of the class Library. The entry points into the class Library typically will 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**

```

BegClass MyJob Extends(ASNA.Monarch.Job) Access(*Public)
    Dc1DB Name(MyDatabase) DBName("**Public/Fig5047") Access(*Public)
EndClass

```

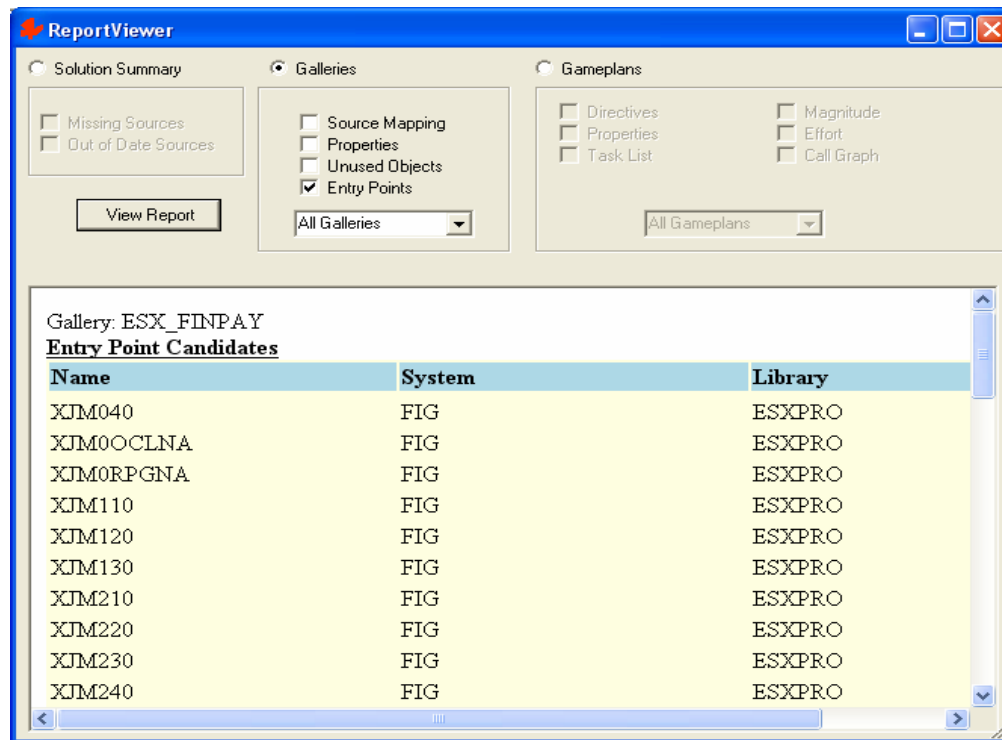
## Finding Entry Points

Deciding how to partition an application is part science and part art. The first thing to know is which Programs are the natural entry points. Monarch Cocoon provides a report to find these Programs.

**To run the report:**

1. Start by selecting the **Report → Migration Effort Assessment**.
2. Next, click the **Galleries** radio button and the **Entry Points** check box.
3. Finally, click in the **View Report** button.

The report will show all those Programs which no other Program calls.

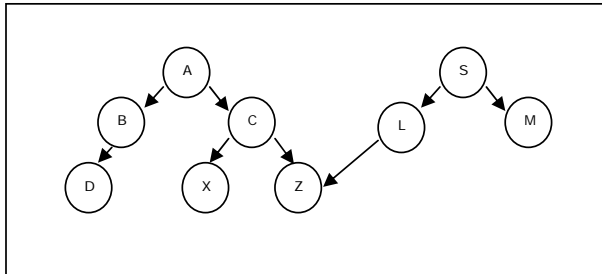


The previous figure shows the report. Of course, any knowledge that the Migrator might have of the application is very useful in determining the optimum partitioning.

If the application is well documented it will also be easier to determine the best partitioning.

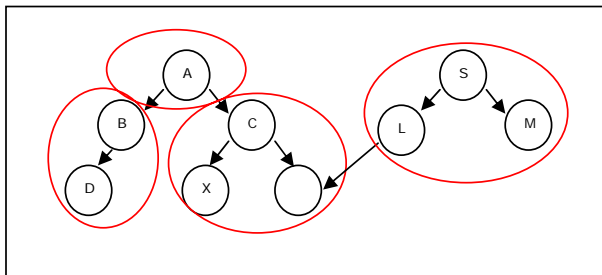
Using the report as an initial list is a good starting point. However the report has a major flaw; it can't account for Programs that are called using a variable name. The report is produced from the information available in the Program itself. This information only registers those Programs that are known to be called at compile-time. CALL statements that use a character variable can be calling any Program, it is not possible to know what the value of the variable will be until the Program executes. This is an area where knowing how the application operates is very useful.

The report is also not useful in determining which Programs should be packaged as a Class Library. Once again, knowledge of the application is required here. Take the following diagram, for example.



Formatted: Centered, Space  
Before: 0 pt, After: 0 pt

This system has two natural entry points, hinting to two applications, one starting at Program **A**, and the other at Program **S**. Taking the Program **A** application in isolation could lead us to any one of the 6 different partitioning in figures 3.X and 3.Y, or some other arbitrary partitioning. However, when we take into account Program **S**, it becomes clear that having Program **Z** in some class Library is desirable. For example the following partitioning would be convenient.

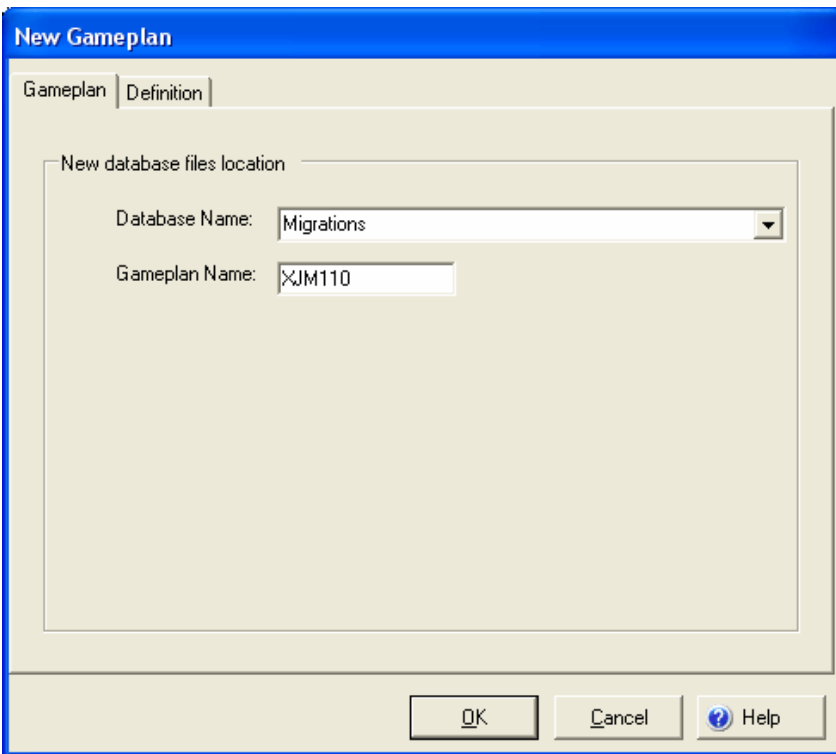


Formatted: Centered, Space  
Before: 0 pt, After: 0 pt

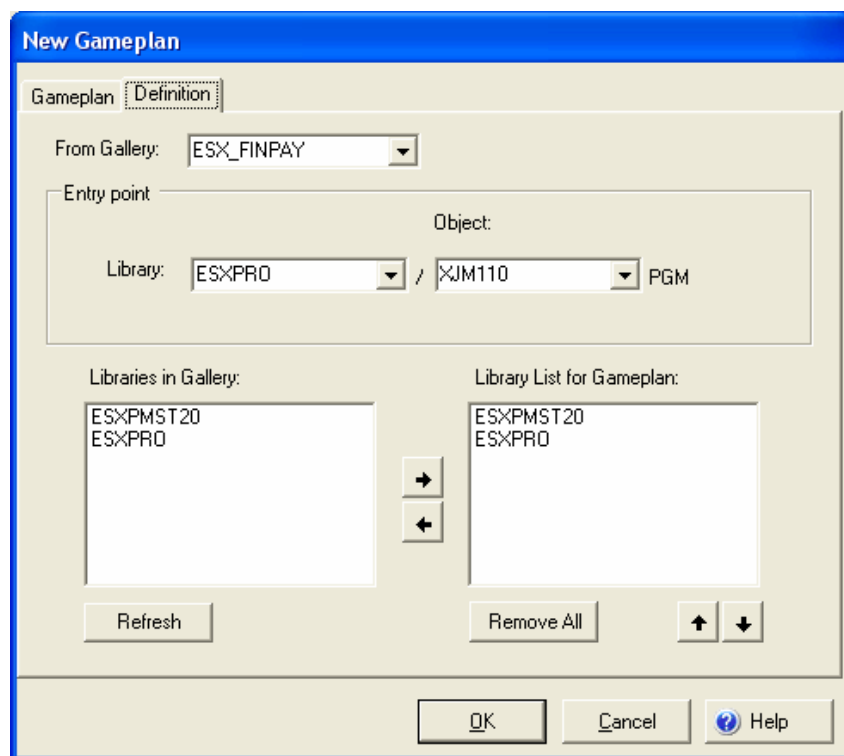
Formatted: Centered, Space  
Before: 0 pt, After: 0 pt

## Creating GamePlans

Once an entry point has been determined, creating a GamePlan is easy, start by selecting the entry point Program within the Gallery and then from the context sensitive menu (right mouse menu) select the → **New GamePlan**. Just like in the creation of a Gallery, the first thing to do is provide a name to the GamePlan and a database name to hold it. The following figure shows the New GamePlan dialog.



The second tab, labeled **Definition**, is used to provide a Library list and make changes to the initial Program selection; as shown below.



Pressing **OK** will start the process of the GamePlan creation. The entry point Program, along with all the Programs it calls and the objects they refer to will be added to the GamePlan.

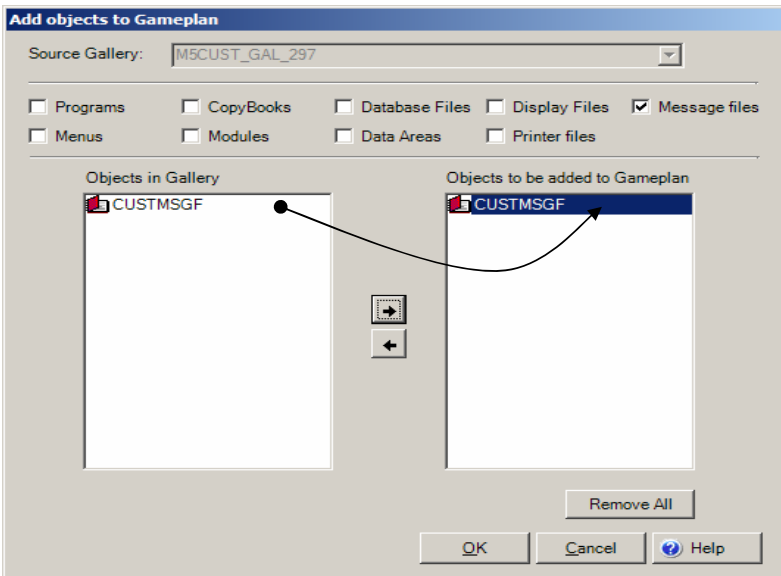
The order of libraries in the Library List for the GamePlan is significant, because when an object is referred to by Library list (\*LIBL) and there are multiple objects with the same name, Cocoon will select the first one found according to the sequence of libraries given. This is especially significant for Program calls. When Cocoon assembles the set of objects to be included in the GamePlan, the Program-call tree is traversed, thus Programs in the tree are those found according to the Library list.

As each object is added to the GamePlan, the Directives established for the object in the Gallery are copied as the initial Directives in the GamePlan. ***It is very convenient to have established the Source Mappings in the Gallery prior to the creation of the GamePlan.*** However, it is certainly possible to edit the mappings directly from the GamePlan.

## Including Additional Objects

When the GamePlan is created, all objects used by the Entry Point Program (and the Programs it calls) are included in the GamePlan. However, there are some object types that can't be determined statically as being used by a particular Program, chief amongst them are Message Files.

If a Message File is used by a single Program, it is convenient to manually add it to the GamePlan where the Program exists. This can be done by selecting any one of the nodes in the GamePlan Solution Explorer window and invoking the context-sensitive menu by clicking the right-mouse button, then selecting the **Add Objects** menu option. The next figure shows the dialog prompting for the objects to be added to the GamePlan.

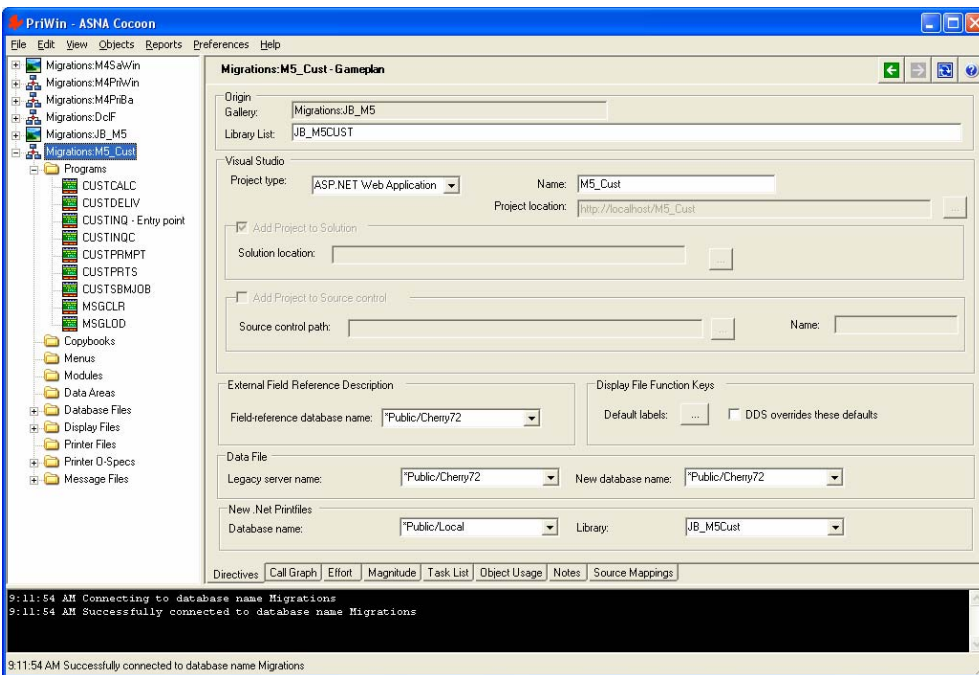


Another case where objects need to be added manually is when creating a GamePlan which will generate a class Library. The class Library might have multiple entry points, but when the GamePlan is created, one entry point can be selected. You can add more Programs to the GamePlan by selecting them manually. When these additional entry point Programs are added, Cocoon will bring into the GamePlan all Programs called by them and the corresponding objects used by the whole lot of Programs.

---

## Establish GamePlan Directives

One of the first activities once the GamePlan is created involves establishing the Directives for the GamePlan. The next figure shows the Directives tab for a GamePlan called **M5\_Cust**.



## Origin

The Directives page shows the parameters that were used when the GamePlan was created, these are the Gallery (Database Name: Gallery Name) and the Library list in effect when the GamePlan was created. This information comes under the Origin group box.

## Visual Studio Options

The next section in the GamePlan Directives page deals with the type of Visual Studio Project that will be generated. As discussed in the Project Types section, there are currently four types available:

1. ASP.NET Web Application
2. ASP.NET Web Class Library
3. Console Application
4. Class Library

If there is at least one display file in the GamePlan, then Cocoon allows only the ASP.NET Project types. Conversely if there are no Display Files included in the GamePlan, you can only choose between Console application and Class Library Project types.

Selecting the name and location for the generated VS Project is dependent upon the type of Project selected. For ASP.NET types, the location will always be set to the **//localhost**. The only option you have is to set the name. For the other Project types, you can select both the name and the location.

The source controls options are there for future releases of Cocoon.

## External Field Reference Description

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) uses the current Library list at compilation time to find the referred file. When these constructs are being migrated, it is necessary for Cocoon to find the reference files. **You must provide a DataGate database name pointing to the server where Cocoon can find the files used as reference files.** This database name must also have its Library list set in a similar fashion as the list would be established at compilation time on the iSeries.

## Display File Function Keys

### Default Labels

When a Display File is migrated, the available Function Keys available at the File or Record label are converted into buttons. The availability of functions 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 the comment was not provided in the DDS source, so there is no way of determining what a good label would be for the buttons, so the function key name is used.

Enter	Print	PgUp	F13	F14	F15	F16	F17	F18	F19	F20	F21	F22	F23	F24
Help	Clear	PgDn	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12

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 comment as labels for function keys.

## Data File

The section deals with DataGate database name pointing where actual data is found, both in the Legacy system as well as the new database server.

The **Legacy server connection** is used to point to the server where the actual application resides. The only usage of this connection, in Monarch 2.0, is to find the Message Files when they are migrated. The Library list on this DataGate name needs to be established so that Message files can be found thru it.

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

## New .NET Print files

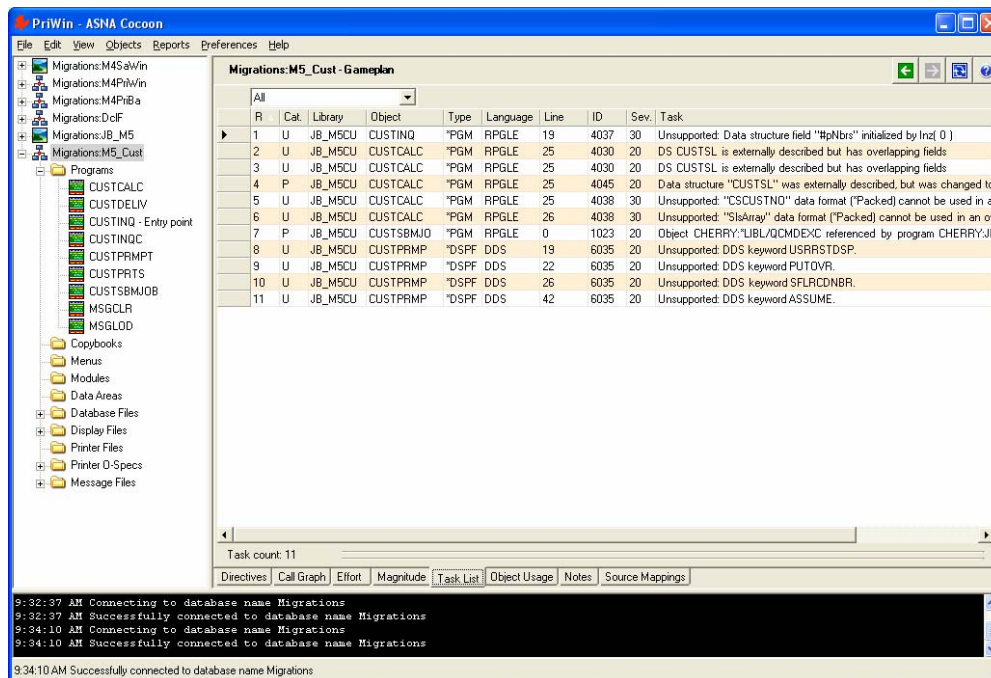
Similar to the database name used for data files, a DataGate database name is needed to locate the migrated print files. This database name is also used when the Print files are being migrated.

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, a Library name can be entered to avoid attempting to create a DataGate print file in the same Library where the legacy OS/400 print file exists, because both kinds of Print files can't 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 to require the Migrators intervention. Simply click on the **Task List** tab of the GamePlan. The next figure shows a task list.



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

**Tasks are organized in the following categories:**

- Problems
- Unsupported Features

- Exceptions

## Problems

This category of tasks involves significant issues which 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 App Name has not been specified.
- Copybook was not found.
- Unauthorized Access while attempting to write,
- Some O-Spec records for {0} 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. Program Directives indicate that the Program is external to this application, yet the "External class name" or "Subroutine name" entries are empty.
- 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. Chapter 11 is devoted to addressing the most common unsupported features and their proposed Solutions. Examples of the kinds of tasks in this category are:

- Program calls with a variable Program name.
- Unsupported CL Command.
- Unsupported Keyword(s) CL Command.
- 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.
- Data structure field is being initialized.
- Unsupported DDS keyword.

## Exceptions

Finally, there are a set of tasks that are issued when Cocoon or one of its agents find an unexpected situation that can't be handled (read as bug).

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

---

## Sizing a Migration

Monarch can assist in the measuring of 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 view Magnitude Density and the second 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 being migrated. It is based mostly in lines of code but it also takes into account other attributes of the sources. Only these objects – which have source lines – 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:

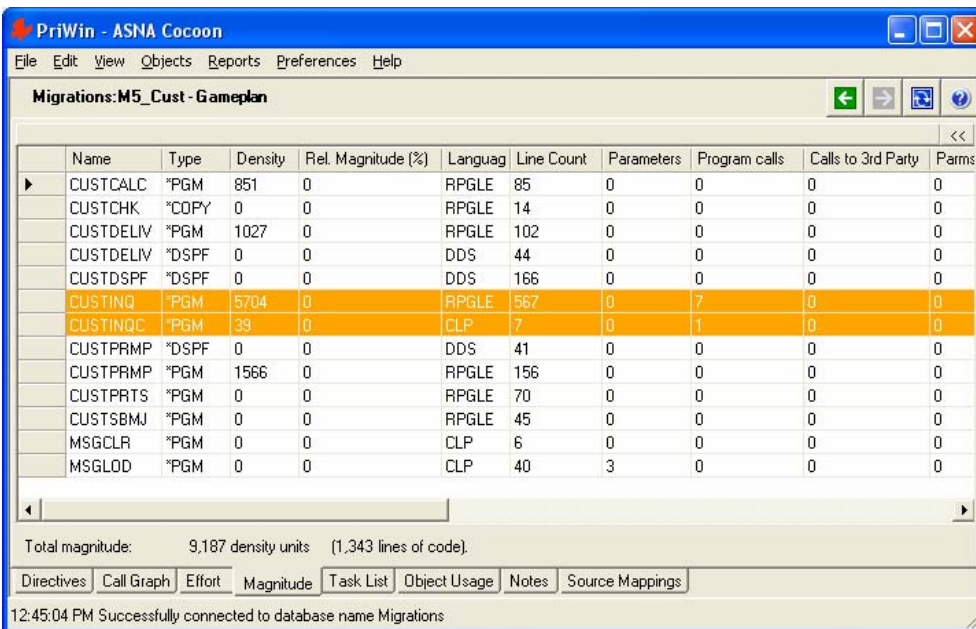
<b><i>Object Type</i></b>	<b><i>Line Cost</i></b>
CL Program	5
RPG/400 Program	1
RPG ILE Program	1
Display File	10
Print File	10

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:

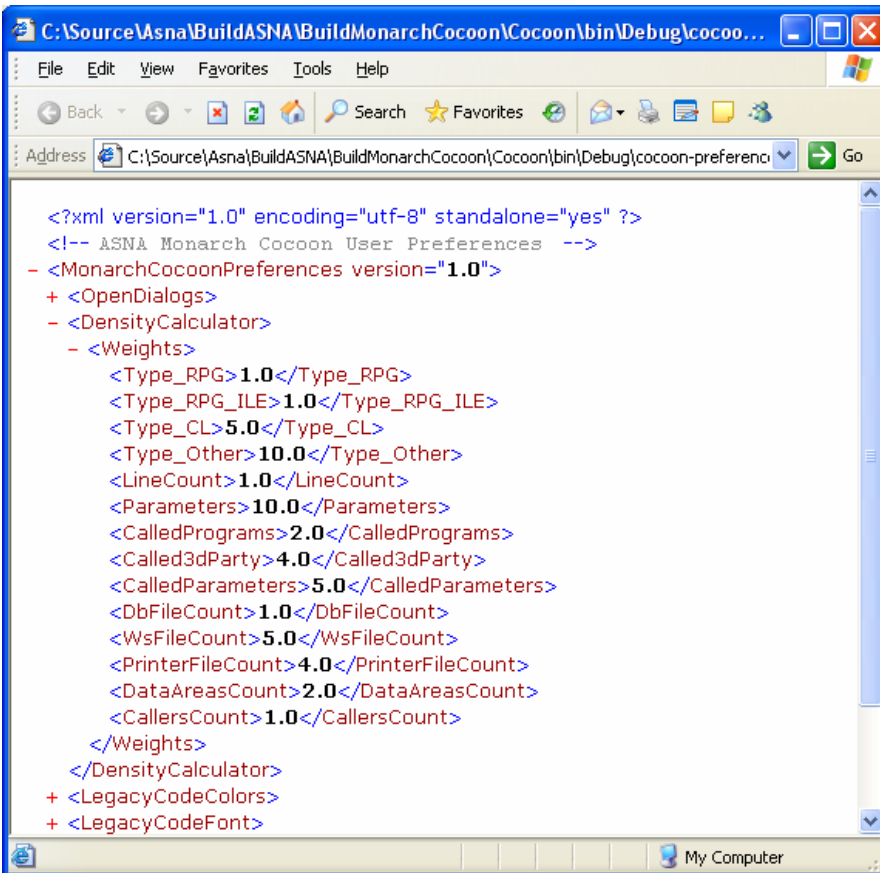
Attribute	Weight
Line Count * Line Cost	1.0
Number of parameters	10.0
Called Programs	2.0
Called Parameters	5.0
Called 3 <sup>rd</sup> Party or unknown programs	4.0
Database Files Used	1.0
Workstation Files Used	5.0
Print File Used	4.0
Data Areas Used	2.0
Programs Calling this one	1.0

If any error is encountered in the process of computing an object's density, the result 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 to the GamePlan. The following figure shows an example.



### 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 in the same folder where cocoon.exe is found, typically: /Program Files/ASNA/Monarch/Cocoon. The file includes a section named **<DensityCalculator>** where the parameters can be modified.

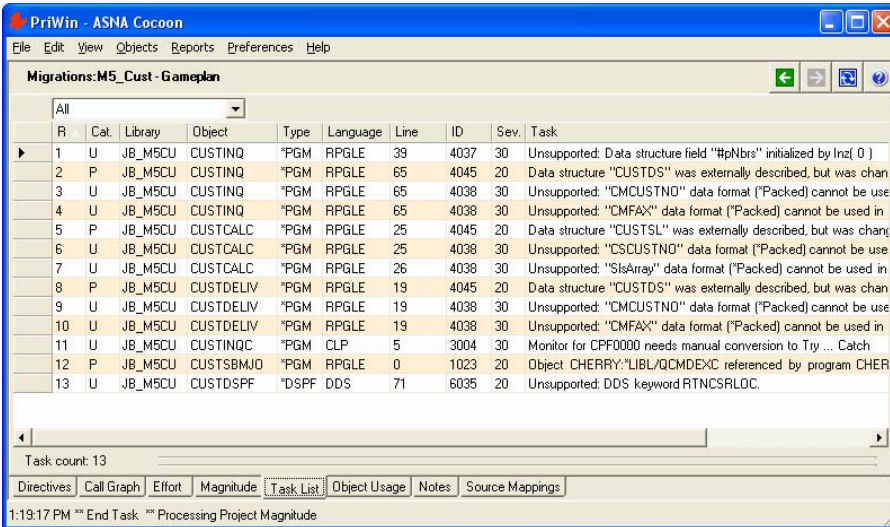


## Effort Density

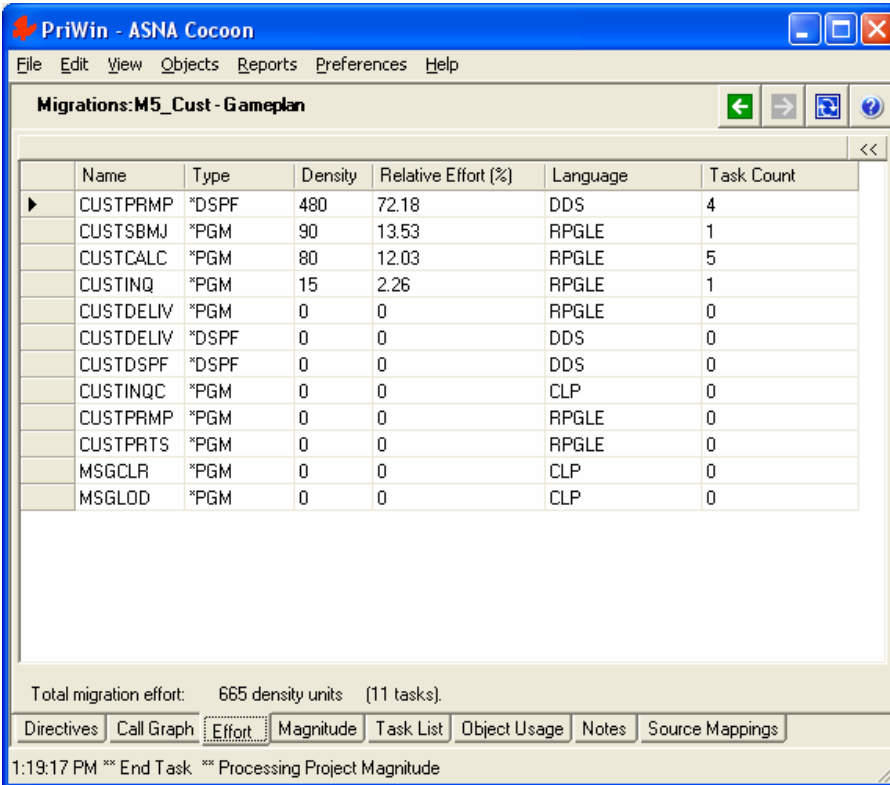
While having an idea of the metrics involving lines of code in a GamePlan is useful in sizing a migration job, it is not enough to have a clear idea of the complete effort required in doing such migration. The Effort Density assigns a number to effort required to migrate individual objects based on the 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.

## Algorithm

Given a list of tasks in a GamePlan like the one show in next figure, a number can be computed for each object to represent the effort required to migrate it.



Each task type has a number assigned representing the cost associated in resolving task. This cost is expressed in minutes. The Effort Density of an object is simply the sum of all the costs associated with each task found in the object. The following figure shows the Effort Density for the task list shown above.



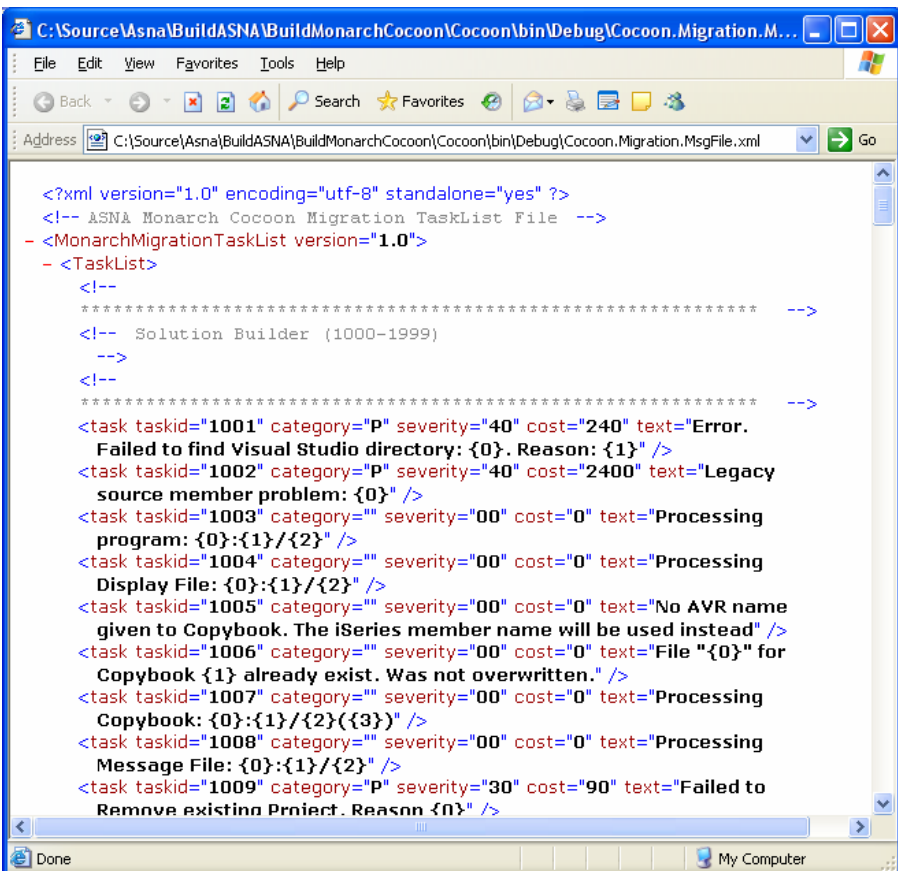
Certain tasks are too generic to have a single cost associated with them, take for instance task **ID 6035: Unsupported DDS Keyword**. Different variations of the task's cost need to be provided with this task, in order for the measurement to be more accurate.

Here is an example of tasks and its corresponding costs:

Task	Description	Variation	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
6010	Option indicators ignored for keyword: {0}		90
6035	Unsupported: DDS keyword {0}	CHGINPDFT	10
		INDARA	20
		MSGLOC	5
6044	Reference Format '{0}' for File '{1}' not found		240

### Customizations

Different 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 found, and tune 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/Cocoon. The file is in XML format, look under the <TaskList> section.



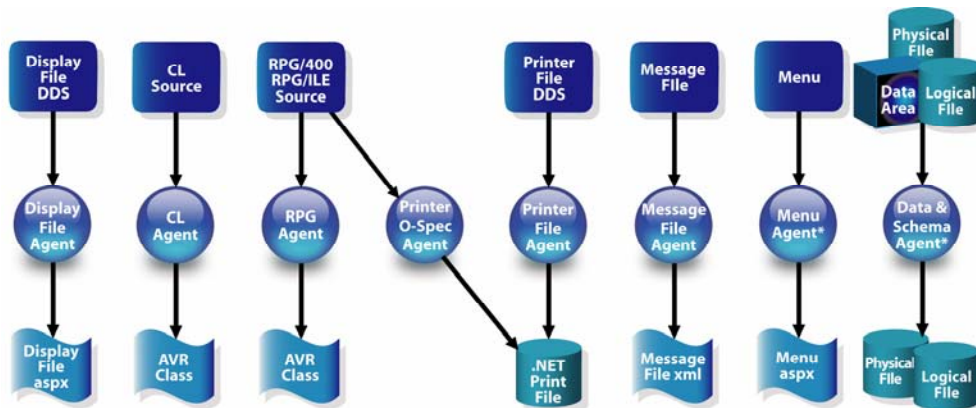
```
<?xml version="1.0" encoding="utf-8" standalone="yes" ?>
<!-- ASNA Monarch Cocoon Migration TaskList File -->
- <MonarchMigrationTaskList version="1.0">
- <TaskList>
  <!--
  ***** -->
  <!-- Solution Builder (1000-1999)
  -->
  <!--
  ***** -->
  <task taskid="1001" category="P" severity="40" cost="240" text="Error.
  Failed to find Visual Studio directory: {0}. Reason: {1}" />
  <task taskid="1002" category="P" severity="40" cost="2400" text="Legacy
  source member problem: {0}" />
  <task taskid="1003" category="" severity="00" cost="0" text="Processing
  program: {0}:{1}/{2}" />
  <task taskid="1004" category="" severity="00" cost="0" text="Processing
  Display File: {0}:{1}/{2}" />
  <task taskid="1005" category="" severity="00" cost="0" text="No AVR name
  given to Copybook. The iSeries member name will be used instead" />
  <task taskid="1006" category="" severity="00" cost="0" text="File "{0}" for
  Copybook {1} already exist. Was not overwritten." />
  <task taskid="1007" category="" severity="00" cost="0" text="Processing
  Copybook: {0}:{1}/{2}{3}" />
  <task taskid="1008" category="" severity="00" cost="0" text="Processing
  Message File: {0}:{1}/{2}" />
  <task taskid="1009" category="P" severity="30" cost="90" text="Failed to
  Remove existing Prniect. Reason {0}" />
```

## Chapter 4 – Pulling the (Migration) Trigger

Executing the migration process is a very simple operation from the Migrators point of view. You simply select the **Object → Migrate** menu option. However, from Cocoon's point of view, the process is very involved. 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 chapter.

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 6 migration agents and one Solution builder. There is a chapter dedicated to each one of these agents. A future release will introduce 2 more agents: Menu Agent and Data & Schema Agent.



Several of these agents operate with artifacts outside the GamePlan itself, especially templates.

### Establishing Templates

It is important to establish the templates for the agents before starting the migration process. Templates influence the output of the agents. If a change is introduced later on in an agent's template, the objects affected by that agent would have to be re-migrated to include the changes in the template.

Subsequent chapters in this guide deal with individual agents. Each one of these chapters will detail what templates are used by the agent and where and how to modify the template.

What is important to know now, is that changing a template does not affect previously migrated objects. Take, for instance, the display file agent. A template determines the look of the ASPX pages generated. It is not uncommon to modify individual pages after they have been migrated.

If a particular common format is desired for these pages, then the template should be established before the migration occurs. Otherwise the migration process will

have to be run again as the Display Files and the individual modifications previously made to the pages would be lost.

## 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 following templates located under the *\$InstalDir\Cocoon\Templates\* folder:

Project Type	Subfolder	Contents
<b>ASP.NET Web App</b>	AspWebApp	!Diagnose.aspx !Diagnose.aspx.vr !EoJ.aspx !EoJ.aspx.vr !ExpiredSession.htm Global.asax.vr MyJob.vr WinPopUp.aspx WinPopUp.aspx.vr
<b>ASP.NET Class Library</b>	AspClassLibrary	!NotCallable.htm Global.asax.vr MyJob.vr
<b>Class Library</b>	ClassLibrary	MyJob.vr
<b>Console Application</b>	ConsoleApp	BatchEntryPoint.vr MyJob.vr

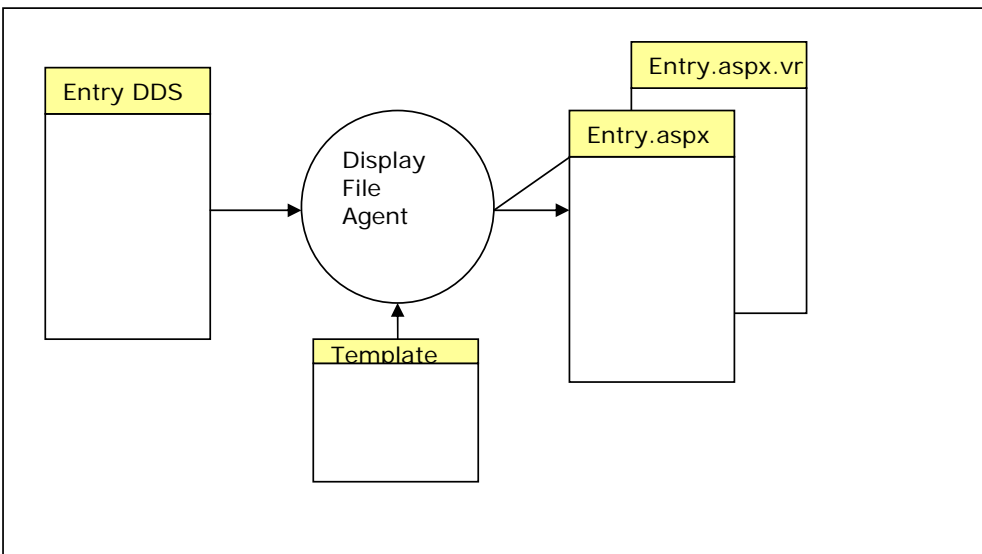
## Chapter 5 – Migrating Display Files

Display files 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.

1. 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.
2. Second, an '.aspx.vr' file contains the code-behind class which 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.

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. Most of the elements found in the DDS specification are migrated. However, there are two general features which today 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 which is converted to a browser pop-up window.

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



### Templates

The Display File Agent uses a template to determine the look of the generated page. The templates are located under the *\$InstalDir\Cocoon\Templates\DisplayFiles* folder. The template is a 'regular' aspx page pair (includes the .aspx and the .aspx.vr source files) with the following requirements:

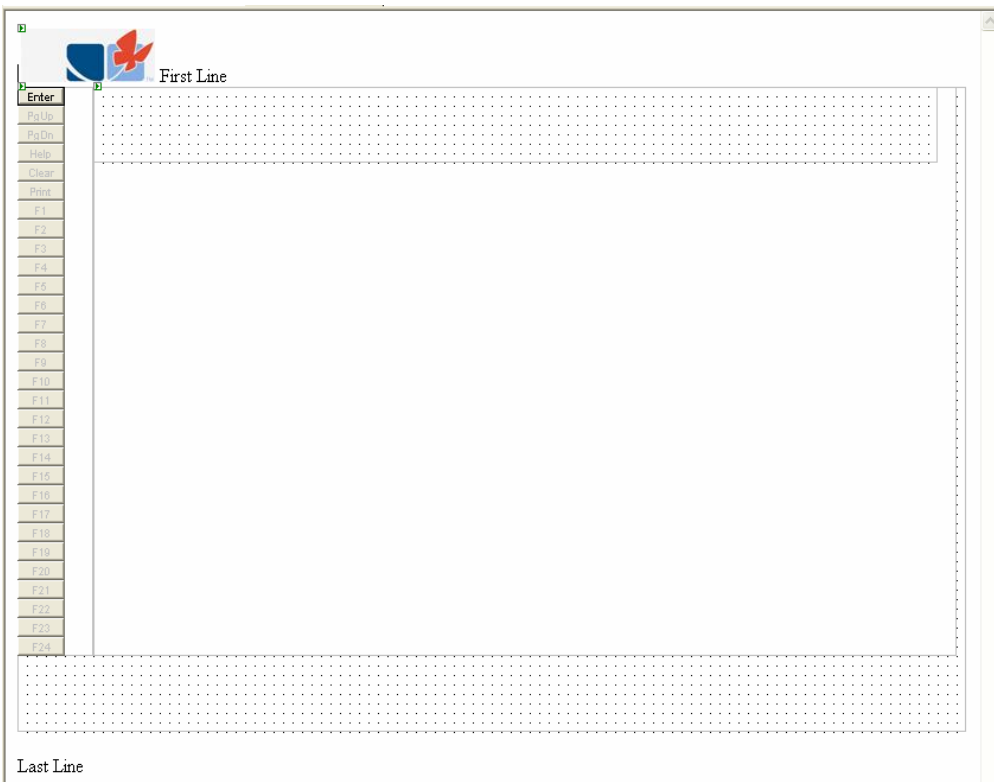
## Template.aspx

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 DdsRecords (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).

## Template.aspx.vr

The Template.aspx.vr file must contain a class which derives from **ASNA.Monarch.WebDspF.Page**. It should include a line at the global level that starts with the command **ImpDspFile**. Typically it will immediately precede the first subroutine **OnInit**. This line will get replaced with the declaration of all DdsFile, DdsRecords, DdsFields. The **ImpDspFile** will also be completed with the detail of the display file. The source may contain declaration for Dds widgets, but they will be removed from the final generated source.

## Example



## Template.aspx

```

<%@ Page language="AVR" Codebehind="DspTemplate.aspx.vr"
    AutoEventWireup="false" Inherits="M5_Cust.DspTemplate" %>
<%@ Register TagPrefix="mdf" Namespace="ASNA.Monarch.WebDspF"
    Assembly="ASNA.Monarch.WebDspF" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<HTML>
  <HEAD>
    <title>DspTemplate</title>
    <meta content="Microsoft Visual Studio 7.0" name="GENERATOR">
    <meta content="AVR" name="CODE_LANGUAGE">
    <meta content="JavaScript" name="vs_defaultClientScript">
  </HEAD>
  <body>
    <form id="Form1" method="post" runat="server">
      <asp:image id="Image1" style="Z-INDEX: 101" runat="server"
        ImageUrl="./Monarch_logo.jpg" Height="45px"
        Width="113px" ImageAlign="Left">
      </asp:image><br>First Line<br><br>
      <DIV style="WIDTH: 800px; POSITION: relative; HEIGHT: 544px"
        ms_positioning="GridLayout">
        <DIV style="Z-INDEX: 102; LEFT: 0px; WIDTH: 64px;
          POSITION: absolute; TOP: 0px; HEIGHT: 480px"
          ms_positioning="FlowLayout">
          <mdf:ddsfile id="DdsFile1" runat="server"
            BannerStyle="Vertical"></mdf:ddsfile>
        </DIV>
        <DIV style="Z-INDEX: 103; LEFT: 64px; WIDTH: 728px;
          POSITION: absolute; TOP: 0px; HEIGHT: 480px"
          ms_positioning="FlowLayout">
          <mdf:ddsrecord id="DdsRecord1" style="POSITION: relative"
            runat="server" Height="64px" Width="712px"
            ms_positioning="GridLayout"></mdf:ddsrecord>
        </DIV>
      </DIV>
      Last Line
    </form>
  </body>
</HTML>

```

## Template.aspx.vr

```

Using System
Using System.Collections
Using System.ComponentModel
Using System.Data
Using System.Drawing
Using System.Web

```

```
Using System.Web.SessionState
Using System.Web.UI

Using System.Web.UI.WebControls
Using System.Web.UI.HtmlControls

BegClass DspTemplate Extends(ASNA.Monarch.WebDspF.Page)
    DclFld DdsFile1 Type(ASNA.Monarch.WebDspF.DdsFile) +
        Access(*Protected) WithEvents(*Yes)
    DclFld DdsRecord1 Type(ASNA.Monarch.WebDspF.DdsRecord) +
        Access(*Protected) WithEvents(*Yes)
    DclFld Image1 Type(System.Web.UI.WebControls.Image) +
        Access(*Protected) WithEvents(*Yes)

    ImpDspFile

    BegSr OnInit Access(*Protected) Modifier(*Overrides)
        DclSrParm Name(e) Type(System.EventArgs)
        //
        // Required by the ASP.NET Web Form Designer.
        //
        InitializeComponent()
        *Base.OnInit(e)
    EndSr

    /region Web Form Designer generated code
    // Required method for Designer support - do not modify
    // the contents of this method with the code editor.
    BegSr InitializeComponent
    EndSr
    /endregion

EndClass
```

## Chapter 6 – Migrating Printer 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. Monarch provides two migration agents to deal with these two techniques. Both agents target the DataGate Printer File facilities. These facilities consist of:

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

The printer O-Spec agent takes the RPG O-Specs as input and generates a DataGate Print File. This agent is detailed in a subsequent chapter.

The print file agent takes DDS specifications as input and creates a DataGate Print File.

## Interesting Keyword translations

### **BARCODE**

Translation of this keyword will consist of setting the font of a 'regular' field to a user-selectable name. No other parameter of this keyword is supported in this release. It is the responsibility of the Migrator to acquire and distribute the fonts used for printing barcodes. There are many companies selling different implementations of bar-coding via fonts.

### **Fonts**

There are 4 keywords that can be used to select a font to be used for printing fields and constants. However, none of these keywords are currently supported.

- CDEFNT - specify the coded font.
- FNTCHRSET - specify the font.
- FONT - specify the font ID.
- FONTNAME - specify the TrueType font name.

## Chapter 7 – Migrating CL Programs

A CL Program on OS/400 has the capability of using over a thousand commands provided by OS/400, 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 doesn't 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 on 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, RETURN
- MONMSG
- ADDLIBLE, RMVLIBLE
- DCLF, RCVF, SNDF, SNDRCVF
- OVRDBF, OVRDSPF, DLTOVR
- CLRPFM, INZMBR
- CHGDTAARA, RTVDTAARA
- RTVJOBA
- SNDPGMMSG, RMVMSG

## Resources

**Alphabetic List** of CL Commands:

<http://publib.boulder.ibm.com/series/v5r2/ic2924/info/rbam6/rbam6alphatable.htm>

**CL Command Finder** page

<http://publib.boulder.ibm.com/series/v5r2/ic2924/info/rbam6/rbam6clfinder.htm>.

See the following resource for more information about using CL. [CL](#)

[Programming](#) 

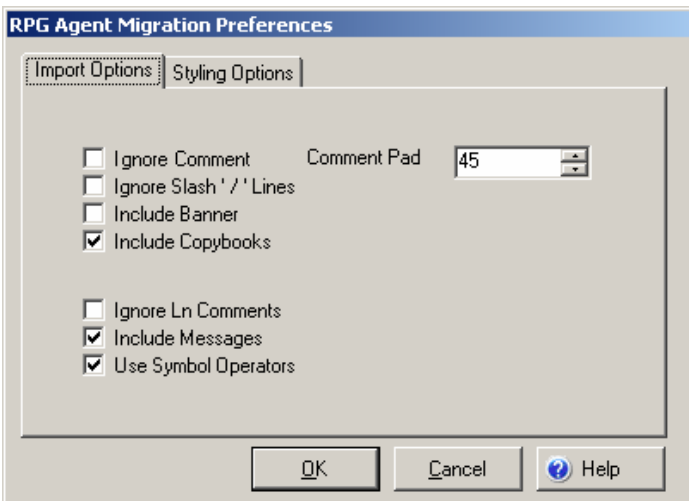
(<http://publib.boulder.ibm.com/series/v5r2/ic2924/books/c4157215.pdf>). This book provides a wide-range discussion of Programming topics, including control language Programming, Programming concepts, objects and libraries, message handling, and user-defined commands and menus.

## Chapter 8 – Migrating RPG Programs

### Setting up Migration Preferences

Before migrating your RPG source code, you may want to adjust the **RPG Migration Preferences** offered by Cocoon.

From the Cocoon Main menu, select the **Preferences** menu, then point to **RPG Agent Migration Preferences**. The RPG Agent Migration Preferences dialog will display, as shown below.



The **“Import Options”** tab shows a group of selections that determine what the RPG Agent will do while parsing legacy source code. For example, should the RPG Agent throw away legacy comment lines before starting the conversion?

The **“Styling options”** tab shows a group of selections that determine what the RPG Agent will do during the generation of the migrated code. For example, when converting a CALL, there are four alternative AVR constructs that are equivalent; it is just a matter of programming style (see Figure on page 55).

### Import Options

#### Ignore Comment

Select this option to have RPG Agent remove comment lines as defined by RPG (a **‘\*’** in position seven). When unchecked, the comment will be preserved in the new AVR code, by preceding it with a **“\”** symbol.

This setting also affects specs that are not valid (like a blank on column seven), or specs that are consumed by AVR, but the actual specification does not have an equivalent op-code in AVR. An example of this is the **‘H’** specs, where the default date and time is changed - and used in the following date/time format op-codes.

Lastly, if any short lines are detected (lines shorted than seven characters) when ignoring comments, the contents of these lines are not used in the new file.

### Ignore Slash '/' Lines

Select this option to have the RPG Agent remove lines that have a slash ('/' character), in position seven. These lines are iSeries-compiler directives, such as: /TITLE, /EJECT, /SPACE. One exception is the /COPY that is always processed by Monarch (see Discovering Copybooks in Chapter 2).

When unchecked, the line will be treated as if there was a '\*' in position seven and then the "Ignore Comment" (above) option applies.

### Include Banner

Select this option to include a "banner" before the body of a subroutine. A banner is a block of text of the form:

```
//-----
//-- subroutine name
//-----
```

### Include Copybooks

*This option has been deprecated. It is no longer used by RPG Agent for copybook handlings see Discovering Copybooks in Chapter 2.*

### Ignore Ln Comments

"Ln comments" (line-comments) are the comments that may be entered in RPG on the same line as the different specs, usually starting at position 80. When unchecked, the new migrated line: 1) gets padded to the right with blanks (see "Comment pad" option, below), 2) then a "/" symbol is appended and 3) lastly, the text comment is transferred from the original line.

### Include Messages

Select this option to include Monarch-generated comment lines, such as:

```
// name -- KLIST was Relocated just before "BEGSR Process_Mainline_Code"
in this Program
```

### Use Symbol Operators

Select this option to have the RPG Agent replace binary compare op-codes embedded in several RPG op-codes for its equivalent symbol in AVR.

When selected, the following table is used:

Last two characters of operator	Replacement symbol
"EQ"	" = "
"NE"	" <> "
"LT"	" < "
"GT"	" > "
"LE"	" <= "

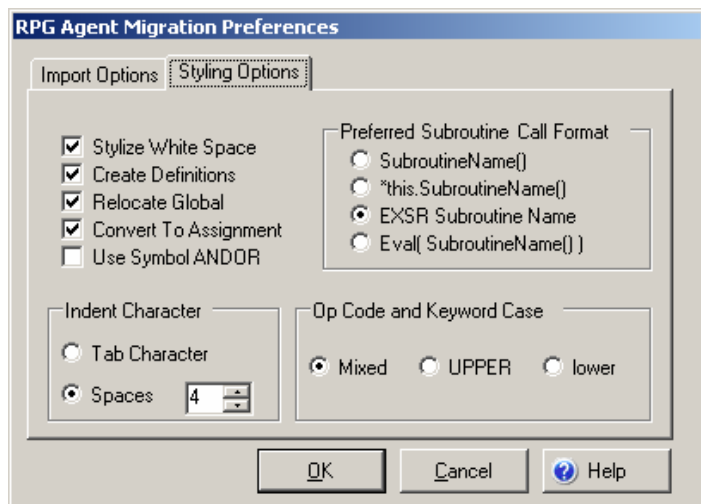
"GE"	" >= "
------	--------

For example, WHENEQ will be matched with "EQ" and the symbol "=" will be used instead; same for WHNE, IFLT, DOWGE, CABLE, CASGT, etc.

**Comment Pad**

When *not* ignoring line comments (see "Ignore Ln Comments" above), this entry allows you to specify the amount of blank spaces to be used while padding the line to the right **before** copying the line comment.

**Styling Options**



*RPG Agent's "Migration Styling" options*

**Stylize White Space**

If this option is unchecked, then before writing a converted line, the RPG Agent looks for "(" and ") ". If found, the following two patterns are replaced by the corresponding white space on the column on the right:

Pattern to be replaced	Replacement value	Comments
"( "	"("	Space after open parenthesis removed.
" ) "	" ) "	Space before closed parenthesis removed and two blanks added after the closed parenthesis.

## Create Definitions

When this option is checked, all field definitions are collected, and a “sorted list” of global declarations is inserted right before the `*Entry` function.

## Relocate Global

*This option should be always checked. Unchecking the option will produce code that does not compile. The intention was to give the migrator the option to relocate Key lists to the global declaration. This is now required by AVR.*

## Convert to Assignment

When this option is checked, any math operation code (`ADD`, `SUB`, `DIV`, `MULT`, `ZADD` and `ZSUB`), will use the equivalent AVR math symbols (`+`, `-`, `*`, `/`).

## Use Symbol ANDOR

When this option is unchecked (default), the following keyword will be used when generating conditional expression code. If checked, then the corresponding symbol is used in the new syntax, as listed below.

Option unchecked	Option checked
*And	&
*Or	
*Not	~

## Indent Character

The two mutually-exclusive options are:

### Tab Character

A single tab is used per indent level.

### Spaces

The number of spaces indicated is used per indent level.

## Preferred Subroutine Call Format

The four mutually-exclusive options are:

SubroutineName()

\*this.SubroutineName()

EXSR Subroutine Name

Eval( SubroutineName() )

## Op Code and Keyword Case

The three mutually-exclusive options are:

### Mixed

A combination of Upper and lower case characters are used for op-code translation. When this option is used, the new op-codes are generating starting with an upper case letter for each word in the op-code. For example: `Dc1f1d`, `DclArray`, `TimFmt`, etc.

### UPPER

Op-codes are converted to all uppercase. For example: `DCLFLD`, `DCLARRAY`, `TIMEFMT`.

### lower

Op-codes are converted to all lowercase. For example: `dc1f1d`, `dclarray`, `timefmt`.

## Program Migration Directives

Each program is associated with a set of migration parameters called **Directives**. The Directives page is the default page that displays when selecting a program from the Solution Explorer tree, as shown below.

The screenshot shows a software interface window titled "Public/My Migrations:CUSTINQC - Gameplan". It features a "Origin" section with two text boxes: "Gallery:" containing "My Migrations:M3SALES\_GAL" and "Library List:" containing "M3SALES". Below this are three radio buttons for "Internal", "Remote", and "External". The "Internal" radio button is selected, and its corresponding section contains a "Class name:" text box with "Custinqc" entered and a checked "Migrate" checkbox. The "Remote" section has "Database Name:", "Library:", and "Program Name:" text boxes. The "External" section has "Class name:", "Subroutine:", "DLL name:" (with a browse button "..."), and "Namespace:" (with an unchecked checkbox) text boxes. At the bottom, there are five tabs: "Directives" (selected), "Reference Diagram", "Legacy Source", "Used By", and "Notes".

## Origin

### Gallery

Shows the fully-qualified name of the gallery used to create the Gameplan where this program exists.

### Library list

Shows the library list to use when resolving `*LIBL` during program-reference lookup.

Initially, the blank-separated list of libraries is set to the list of the libraries discovered while creating the gallery. However, this list may be later edited when the Gameplan node is selected, thus becoming a true wildcard search list (for program-reference lookups).

**Note:** Other agents use library lists that are defined by the database connection, such as the Display file Agent, which uses the library list associated with the database connection specified in the "External Field reference Description" section of the GamePlan's Directives.

### Migrate checkbox

When checked, the program becomes part of the list of programs to migrate when the whole Gameplan is migrated in batch.

## Internal

When the application calls this program *anywhere* in the code, this option directs the Agent to produce an internal call. This gets translated to an invocation of the class that implements the program and a call to the `Process_Mainline_Code()` method.

### Class name

When the program is internal to the application, the name of the class that implements it in the new code may be changed. It defaults to the name of the iSeries object. The length of the name is no longer limited to ten characters as in the iSeries. 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.

## Remote

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 iSeries.

### Database

When the program stays on the iSeries, Monarch needs the name of the database connection to construct the remote call everywhere any program refers to this one.

### Library

When the program stays on the iSeries, Monarch needs the name of the iSeries library where the program exists.

### Program name

Enter the name of the iSeries **PGM** object to be used in all remote calls to this program throughout the application.

## External

**When a program is external to the application, Monarch needs to load a .Net library (equivalent to service program on the iSeries), locate the class, instantiate it and then call one of its methods.**

**Class name**

Enter the name of the class that represents the program to be called. Usually the name of the class is the name of the iSeries PGM object, but it may have been changed by the Migrator while working with the GamePlan that produced the “Class Library” (DLL).

**Subroutine**

Enter the name of the method (in the class) to be called everywhere this program is referenced throughout the application. Usually the name of the subroutine is \*Entry, but when calling any other .Net Class Libraries (not necessarily written in AVR), the name is the name given to the method given when implementing the class.

**DLL name**

Enter the name of the .Net assembly (Project name, when this program was a Monarch generated Class Library Gameplan type). When calling third party DLLs (service programs), enter the name of the .Net assembly filename.

The name entered may include the full search path, if the assembly resides in a particular folder in the deployment server, or it may be just a name, when the assembly is to be found in the GAC (Global Assembly Cache).

---

## Op-Code Support

All RPG Op-codes are supported except for the following:

**ACQ, ALLOC, DEALLOC, DEBUG, DSPLY, DUMP, FORCE, NEXT, POST, REALLOC, REL, RESET** and **SHTDN** (**CALLP** and **EVALR** are being added to the next Monarch service release).

### Normalization of RPG Op-codes

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

Deprecated name	Normalized Name
BITOF	BITOFF
CALLB	CALL
CAT	CONCAT
CHEKR	CHECKR
COMIT	COMMIT
COMP	COMPARE
DEFN	DEFINE
EXTRCT	EXTRACT
EXCPT	EXCEPT
DELET	DELETE

DOU	DOUNTIL
DOW	DOWHILE
ITER	ITERATE
LOKUP	LOOKUP
MOVEA	MOVEARR
MVR	MOVEREM
PARM	DCLPARM
OCUR	OCCUR
REDPE	READPE
RETRN	RETURN
ROLBK	ROLLBACK
SELEC	SELECT
SORTA	SORTARR
SUBST	SUBSTR
TEST	TESTTIME
TESTB	TESTBITS
TESTN	TESTNUM
UNLCK	UNLOCK
UPDAT	UPDATE
Z-ADD	ZADD
Z-SUB	ZSUB

---

## RPG Compiler Directives

Most RPG compiler directives (like `/TITLE`, `/EJECT`, `/SPACE`) are ignored by the RPG Agent (either thrown away or converted to an AVR line comment, see *Import Options* above). The most notable exception is `/COPY`. This directive is referred in this book as **Copybook** and is given special treatment (see *Discovering Copybooks* on Chapter 3).

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. Each Copybook Directive may specify whether it should be placed in-line during the new code generation, or whether just a reference should be placed (to keep it externally defined). Regardless of that setting, all Copybooks are read into memory when the RPG Agent parses the program's source code.

## Loading Program References

The first thing the RPG Agent does when migrating a program is to load the program references. The program references are:

1. List of Programs it calls.
2. List of Files it uses.
3. List of Modules it uses.<sup>2</sup>
4. List of Menus it uses.<sup>3</sup>
5. List of Data Areas it uses.
6. List of Copybooks it references.
7. List of Third Party iSeries objects it calls.
8. List of Message Files it references.

If any of the program calls are done thru a variable (\*VARIABLE), then a **Task** is reported. Variable program calls are not supported by Monarch. Monarch needs to find the Directive page of each program it calls in order to have all the information about how to generate the new code to execute the program. Most of the programs end up being internal calls to a *.Net class*.

**Note:** *The current release of Monarch uses exclusively early-binding, which requires the name of the program at compile time (before the program gets executed).*

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 by the migrator. Most of the resolution of the 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 next to the **/Error** directive, analyze it, design the code that needs to be added/changed, **remove** the **/Error** directive line and then re-compile the application.

---

<sup>2</sup> The migration of Module objects is not part of the current Monarch release.

<sup>3</sup> The migration of Menu objects is not part of the current Monarch release.

## Migrated Visual RPG File's Header

```
// Migrated on 8/30/2005 at 3:56 PM by ASNA Monarch(R) version 2.0.306.0
Using System
Using ASNA.Monarch.Program
```

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

Immediately following, a list of `Using` directives will follow. By default, the two `Using` directives will be `System` and `ASNA.Monarch.Program`. The default `Using` directives used are required to avoid fully qualified method calls.

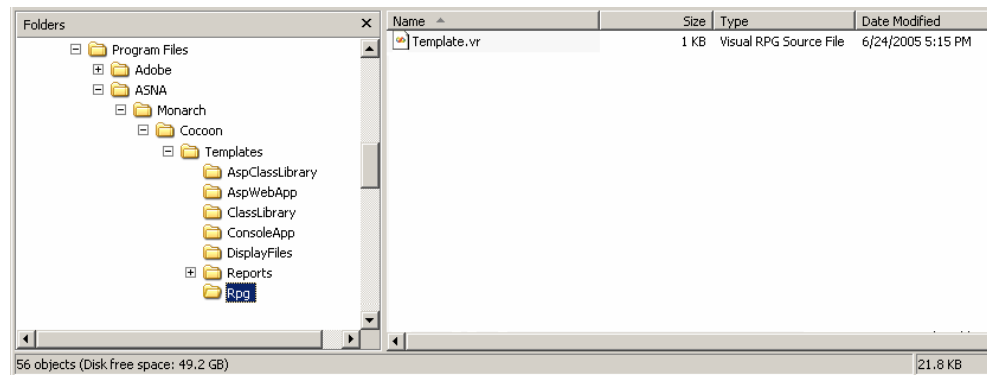
Without `Using ASNA.Monarch.Program`, a line such as:

```
DsLoad TheLda DataArea_In('*LIBL','TheLda',*False)
```

would have to be written as:

```
DsLoad TheLda ASNA.Monarch.Program.DataArea_In('*LIBL','TheLda',*False)
```

If the Migrator wishes to modify the `Using` list, (for example to add more directives at the header of each migrated program) the RPG template file may be modified. The RPG template is a text file located at:



If the RPG template is modified, Cocoon needs to be re-started in order for the RPG Agent to take the new listing.

## New RPG Program's Wrapper Class

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

  BegFunc *Entry Shared(*Yes) Type(*Ind) Access( *Public )
    DclSrParm ...
    DclSrParm ...
    DclSrParm ...

  EndFunc
```

```

BegConstructor Access( *Public )
.
.
.
EndConstructor

BegSr Dispose Access(*Public) Modifier(*Overrides)
.
.
.
*Base.Dispose(Disposing)
EndSr

BegSr Process_Mainline_Code Access( *Private )
//
// 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.
//
EndSr

EndClass

```

### \* Every Migrated Program becomes a Class

All programs using 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 function* 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 \*Entry Function

This function is possibly the most important design of the Monarch framework. The \*Entry function makes the conversion between procedural RPG and an Object-Oriented Visual RPG, preserving the legacy code as close to the original.

The code generated by Monarch to implement the \*Entry function should not be changed, except for possibly adjusting the parameter list to resolve compiler errors or to improve performance or maintainability.

The \*Entry function implements the Activation and Invocation of programs just like on the iSeries. If a program leaves \*INLR set when it completes executing the `Process_Mainline_Code` subroutine (formerly the C-Specs), 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*.

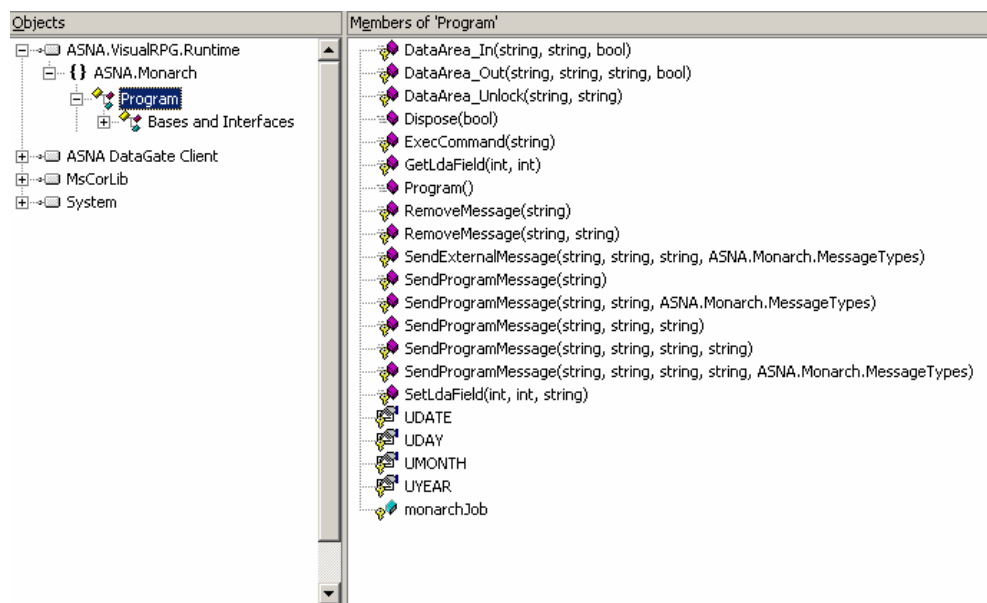
Although is not recommended to change the \*Entry function code, it is important to have a good understanding about the logic behind it, especially during debugging sessions.

The \*Entry function algorithm performs the following:

1. Receive program parameters.
2. Find an instance of this program in memory.
3. If an instance of this program does not exist, then instantiate a new instance.
4. Notify the Job about the invocation of this program (Job adds it to a list)
5. Copy the parameters to the program.
6. Execute the subroutine `Process_Mainline_Code`.
7. Upon returning from subroutine, update the parameters (that may have been modified by the program).
8. Notify the Job about the termination of the invocation (Job removes it from a list)
9. If the \*INLR is set, **destroy** the instance of this program.
10. Return the state of \*INLR to the caller

### ASNA.Monarch.Program Base Class

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



*ASNA.Monarch.Program Interface*

As you can see in the figure above, 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 iSeries).
4. Access to **UPDATE**, **UDAY**, **UMONTH** and **UYEAR**.
5. 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 (described below).

When resolving unsupported features or when modernizing the application once it is migrated, the subroutines and fields defined in ASNA.Monarch.Program are accessible to the migrator. 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 thru **MonarchJob**. The `MonarchJob` class implements the support for the JOB on the iSeries. If you look at the `ASNA.Monarch.Program` figure on page 64, it defines a field called `MonarchJob`, which is an instance to **ASNA.Monarch.Job**.

The Figure on the following page shows a partial listing of the `ASNA.Monarch.Job` members. Your migrated program will have references to some of these members, as you can see in the following code snippet:

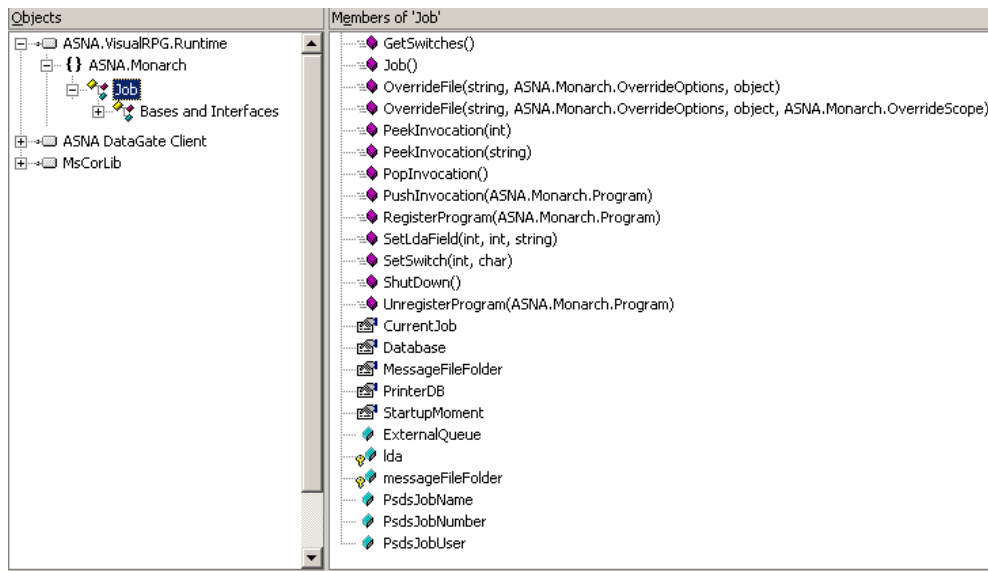
```

BegConstructor      Access(*Public)

    GL0001L.Overrides = MonarchJob
    Open GL0001L DB(MonarchJob.Database)
    QPRINT.Printer = "HP LaserJet 1100 (MS)"
    Open QPRINT DB(MonarchJob.PrinterDB)

EndConstructor

```



Partial `ASNA.Monarch.Job` interface

## Monarch-Generated Properties

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

For migration purposes, properties accomplish the following desirable goals:

### 1. Allow implementation of read-only fields.

For example, it is desirable that `*INLR` is made public, but only for reading, to guarantee that the new state is only changed by the program whose `Process_Mainline_Code` has executed.

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

```

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

```

## 2. 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**, namely: **W9DEBT** and **W9CRED**.

In the code, each field is used individually to move data back and forth.

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 MonarchName0
  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:

```

DclDS MonarchNameO
  DclDsFld #JSCDS          Type( *Char      ) Len( 22 )
  DclAlias W9DEBT Type( *Zoned      ) Len( 11,2 )
  DclAlias W9CRED Type( *Zoned      ) Len( 11,2 )

```

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

### 3. Improve code readability.

In several places, Monarch will make use of properties to simplify fully-qualified names with a shorter (more readable) name. For example, once the property code is added to the program (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 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 thru **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)

```

### iSeries 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 iSeries.

If the migrator needs to move the Data Areas from the iSeries, 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 iSeries Data Area support (ASNA.Monarch.CLProgram provides additional support):

```

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 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          and
SetLdaField
```

**To:**

```
DataArea_In( '', '*LDA', *False)          and
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 gets 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
    CENTRY1 = DataArea_In('*LIBL','CENTRY1',WithLock)
    CENTRY2 = DataArea_In('*LIBL','CENTRY2',WithLock)
    CENTYR = DataArea_In('*LIBL','CENTYR',WithLock)
  EndSr
/endregion

/region ChangeAll Data Areas
  BegSr ChangeAllDataAreas
    DclSrParm WithLock *Boolean
    DataArea_Out('*LIBL','CENTRY1',CENTRY1, WithLock)
    DataArea_Out('*LIBL','CENTRY2',CENTRY2, 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.

### Status Data Structures

The RPG Agent generates a region at the end of every migrated program with several read-only properties to support:

1. Program Status Data Structure.

## 2. File Information Data Structure.

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

The properties generated are only those that the particular migrated program refers to. The name of the property is the name of the structure field defined by the D-Specs.

For example if a program defines **PgmUser** as offset 254 on the Program Status Data Structure,

```
D ProgramDS      SDS
D  PgmUser      254  263
```

Then Monarch generates the following property:

```
/region Status Data Structures
  BegProp PgmUser Type(*Char) Len(10)
    BegGet
      LeaveSr MonarchJob.PsdsJobUser
    EndGet
  EndProp
/endregion
```

Likewise, if a program defines **RecFmtName** as the \*RECORD File Information Data Structure,

```
D WsInfds      DS
D  RecFmtName  *Record
```

Then Monarch generates the following property:

```
/region Status Data Structures
  BegProp RecFmtName Type(*String)
    BegGet
      LeaveSr WorkStationFileName.FormatName
    EndGet

    BegSet
    EndSet

  EndProp
/endregion
```

Notice some interesting details in the implementation. Most of the properties resolve to return the value of a field of the Job or of a Workstation file. In some cases – like in the case for **RecFmtName** – the “Set” accessor is defined, but its implementation is empty. This is necessary to satisfy the compiler, where in some place, the name is used in an expression. An empty set does no harm, and keeps the internal field definition as read-only.

The following table lists the supported Status Data fields:

### Program Status

Offset/Keyword	Description
1	Program
244	Job
254	User
264	Job Number
358	Current User
*PROC	Program
*PROGRAM	Program

### File Status

Offset/Keyword	Description
261	Record
367	Flags
369	AID Byte
370	Cursor Row
371	Cursor Column
397	Relative Record Number
*FILE	File
*RECORD	Record Name

---

## Array Translation

RPG arrays go through two basic conversions:

1. Visual RPG uses brackets ('[' and ']') to access array elements instead of parenthesis.
2. 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
---	--------	--------	------------	----

Gets migrated as:

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

Notice the changes. 1) `SArrayi(3)` became `SArrayi[2]`, 2) The parenthesis was replaced by brackets, and 3) the numeric indexed was decremented by one. If the index was 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 that the migrated program (class) does, when using compile-time data arrays, is to call one subroutine per array to load its data.

The naming convention used for these Monarch0-generated subroutines is:

<code>Load_CompilEtime_Table_ArrayName</code>
---

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:

<pre> 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 </pre>
---

## Generated compile-time loading subroutines

Continuing with the example above, 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[7] = "H"
    LTR[8] = "I"
    LTR[9] = "J"
    .
    .
    .
    LTR[17] = "R"
    LTR[18] = "S"
    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
<code>*** "</code>	Two asterisks on positions 1 and 2 and a blank in position 3.  <i>Note:</i> The data for the arrays must be specified in the same order in which they are specified in the Definition specifications.
<code>***CTDATA arrayname"</code>	Two asterisks on positions 1 and 2 and a blank in position 3, plus the name of the array.  If the <i>arrayname</i> has more than one name (separated with blanks or commas), only the first name is used.  <i>Note:</i> The data for the array may be specified anywhere in the compile-time data section.

## Other Considerations

### Cycle

No current plans are in place to support the Cycle.

There are two possible uses of “The Cycle” in an OS/400 application:

- *Heavy use of the “Cycle”*. If an application relies on the “Cycle” too much, then it would be too complex for Monarch to provide a good translation for that code. The portion of the code that relies on the Cycle will have to be re-written manually using AVR. Matching records is an example of heavy use.
- *Light use of the “Cycle”*. If an application does not rely on the “Cycle” too much, then it will be very easy for the Migrator to translate those few lines manually using AVR code.

### Embedded SQL

Embedded SQL will be supported in a future release. Currently, any embedded SQL has to be translated by hand into ADO. See the *Resolving Unsupported Features* chapter for an example of this translation.

### Advanced RPG ILE considerations

Monarch 2.0 supports only single module ILE RPG Programs. Future Monarch releases will deal with more advanced ILE features including service Programs, multi-module and procedures.

### Internally-Described Files

Monarch supports Program-described Printer Files and its corresponding **EXCPTs**; this support is described in detail in chapter 10.

Database Program-described files are not currently supported by Monarch. The Migrator needs to obtain properly 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 which 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-specs.

## Chapter 9 – Advanced RPG Data Structure Migration

### Overlapping Data Structures

The overlapping Data Structure field feature of RPG is 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 used in a very risky manner, but there are certain common usages that are convenient and valid.

### Data Structures with Gaps

Legacy RPG allowed 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

Where position **three** and position **six** are not used. The legacy programmer most likely assumes such mapping later in the code. For example, mapping the Data Structure to an existing Data Area where those positions are used for other purposes, and when read-in, the values need to go to the proper fields.

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 )      // Filler: 3 - 3
DclDsFld DD          Type( *Zoned ) Len( 2,0 )
DclDsFld MonarchName29 Type( *Char ) Len( 1 )      // Filler: 6 - 6
DclDsFld YY          Type( *Zoned ) Len( 4,0 )
```

A comment to the right of the new unique name-generated field is added, to make it more explicit and assist with maintenance.

### 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 CSMMASTER file has the following definition:

RCSMASTER		Customer Sales and Returns	
Field Definitions			
█ CSCUSTNO	Packed(9,0)	Customer Number	
█ CSYEAR	Zoned(4,0)	Sales Year	
█ CSTYPE	Zoned(1,0)	1 = sales 2 = returns	
█ CSSALES01	Packed(11,2)	Sales Month 01	
█ CSSALES02	Packed(11,2)	Sales Month 02	
█ CSSALES03	Packed(11,2)	Sales Month 03	
█ CSSALES04	Packed(11,2)	Sales Month 04	
█ CSSALES05	Packed(11,2)	Sales Month 05	
█ CSSALES06	Packed(11,2)	Sales Month 06	
█ CSSALES07	Packed(11,2)	Sales Month 07	
█ CSSALES08	Packed(11,2)	Sales Month 08	
█ CSSALES09	Packed(11,2)	Sales Month 09	
█ CSSALES10	Packed(11,2)	Sales Month 10	
█ CSSALES11	Packed(11,2)	Sales Month 11	
█ CSSALES12	Packed(11,2)	Sales Month 12	
Key Definitions			

The field `SlsArray` overlaps the fields: `CSSALES01` thru `CSSALES12`. The intention is to be able to refer to either the whole twelve-packed numbers (year's sales) as a block, for example:

C	xFoot	SlsArray	TempAmt
---	-------	----------	---------

Or to refer to one of the month's sales thru a field name, such as `CSSALES06`. Effectively, the desired behavior is to make `SlsArray(6)`, an element of an array, 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 the new Visual RPG commands, namely `DclOverlayGroup` to produce the following migrated code:

```
// DS CUSTSL has overlapping fields
DclDS CUSTSL // Data structure "CUSTSL" was externally described, but was
  changed to internally described. 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)

```

The new `DclOverlayGroup` maps the `SLSARRAY` array 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), then write it back when the program ends (dispose).

Even when, in this example, the Data Structure does not have overlapping fields, it shows the technique of masquerading the whole Data Structure as a large character field.

Let's 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 iSeries Data Area object.

## Partitioning a Data Structure fields 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 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:

```
// DS MonarchName3 has overlapping fields
DclDS MonarchName3
  DclDsFld DATEU                Type(*Zoned ) Len(8,0)
  DclAlias UC Type(*Zoned ) Len(2,0)
  DclAlias UY Type(*Zoned ) Len(2,0)
  DclAlias UM Type(*Zoned ) Len(2,0)
  DclAlias UD Type(*Zoned ) Len(2,0)
```

The first thing to notice is that Monarch has created a name for the data structure: **MonarchName3**. Modern languages require that all Data Structures are named. A Data Structure in object-oriented languages, such as Visual RPG, are nothing more than a data-only class.

Also, notice the gray boxes shown in the figure. These are collapsed *regions* that Monarch inserts into the migrated code in order to make it more readable.

By expanding the first region, the underlying code can be appreciated, and the technique explained:

```
/Region DclAlias UC Type(*Zoned ) Len(2,0)
  DclAlias UC Exp(Prop_UC) Type(*Zoned ) Len(2,0)

  BegProp Prop_UC Type(*Zoned ) Len(2,0)
  BegGet
    Move DATEU _DATEU Type(*Char) Len(8)
    LeaveSR _DATEU.Substring(0,2)
  EndGet

  BegSet
    Move DATEU _DATEU Type(*Char) Len(8)
```

```

        Move      *PropVal  _UC Type(*Char) Len(2)
        DATEU =  _UC + _DATEU.Substring(2)
    EndSet
EndProp
/EndRegion

```

Monarch makes use of properties to take pieces out of a field. In this case, a property named `Prop_UC` has been created, and the two *accessors* "get" and "set" have been implemented.

A simple code analysis of the implementation of `Prop_UC` shows that each time the code uses `Prop_UC` to retrieve its value, a temporary field `_DATEU` type `*Char`, length `8` is created and the contents of the field `DATEU` are copied into it. Only the first two chars are returned to the caller. In the same fashion, every time `Prop_UC` is assigned a new value in the code, again a temporary field `_DATEU` type `*Char`, length `8` is created, but this time its contents take the new value being assigned, and the two first characters are assembled back into the field `DATEU`.

The effect is *almost* what the legacy code intended. However, there is one extra step Monarch takes to reach the expected result. The legacy code refers to `UC`, not `Prop_UC`. To complete the technique, Monarch makes use of the `DclAlias` Visual RPG command.

The `DclAlias` Visual RPG command allows referring to a symbol using an *alias* name. In this case `UC` is the name we need to give to the property `Prop_UC` just defined.

Anywhere in the code where `UC` is used (the *century* part of the Date), whether to retrieve the value or to assign a new value; the field `DATEU` will be updated, thanks to the property's implementation. The `UY`, `UM` and `UD` (year, month and day) are implemented by Monarch using the same technique used for `UC`.

## Partitioning a Data Structure Fields 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. 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, this overlapping Data Structure is unnamed and the intention is to use the whole account code thru `$glal#` or its parts: `$org`, `$glno` and `$loc` (sharing the same memory).

Monarch produces the equivalent migrated code:

```
// DS MonarchName89 has overlapping fields
DclDS MonarchName89
  DclDsFld $glal#           Type( *Zoned ) Len( 20,0 )
  DclAlias $org Type( *Zoned ) Len( 6,0 )
  DclAlias $gino Type( *Zoned ) Len( 10,4 )
  DclAlias $loc Type( *Zoned ) Len( 4,0 )
```

Looking closely at the implementation of one of the aliases - **\$loc** -, we can see:

```
/Region DclAlias $loc Type( *Zoned ) Len( 4,0 )
  DclAlias $loc Exp(Prop_$loc) Type( *Zoned ) Len( 4,0 )

  BegProp Prop_$loc Type( *Zoned ) Len( 4,0 )
    BegGet
      Move    $glal# _$glal# Type(*Char) Len(20)
      LeaveSR _$glal#.Substring(16,4)
    EndGet

    BegSet
      Move    $glal# _$glal# Type(*Char) Len(20)
      Move    *PropVal _$loc Type(*Char) Len(4)
      $glal# = _$glal#.Substring(0,16) + _$loc
    EndSet
  EndProp
/EndRegion
```

Again, each time the *alias* name **\$loc** is used or assigned to a new value, the **\$glal#** field is modified on his behalf. This is done by breaking **\$glal#** into its components to get the current value, or assembling **\$glal#** from its components.

## Renaming fields and mapping to arrays

Consider the following array declaration and its corresponding Input specifications:

D MSC	S	8	DIM(5)
<b>IMSHMSTRR</b>			
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 defined by:

MSHMSTRR		
Field Definitions		
MRBRN	Zoned(3,0)	[Reference to R_BRANCH]
MRTYCD	Zoned(4,0)	[Reference to R_MSHTYPCD]
MRLNGT	Zoned(2,0)	Length of Membership
MRMAG	Zoned(2,0)	[Reference to R_AGE] Minimum Age
MRMXAG	Zoned(2,0)	[Reference to R_AGE] Maximum Age
MRSEX	Char(1)	[Reference to R_GENDER] Gender
...		
MRMTOD	Zoned(2,0)	Months to Defer
MRDFBS	Char(1)	[Reference to R_FLAG]
MRMSC1	Char(8)	Miscellaneous Charge Code 1
MRMSC2	Char(8)	Miscellaneous Charge Code 2
MRMSC3	Char(8)	Miscellaneous Charge Code 3
MRMSC4	Char(8)	Miscellaneous Charge Code 4
MRMSC5	Char(8)	Miscellaneous Charge Code 5
MRBDCH	Char(6)	Miscellaneous Charge Code 5
MRDEPT	Packed(11,2)	[Reference to R_AMOUNT]
MRUSR	Char(10)	[Reference to R_USERID] User ID
MRUPDD	Date(USA)	[Reference to R_DATE] Date of Last Update
...		

The C-Specs code will use individual elements of the array **MSC**, to refer to the fields **MRMSC1 ... MRMSC5** of the record format **MSHMSTRR**, for example in this Do loop in the following code snippet:

```

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

Monarch's 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 new 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`, `MRMSC4` 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:

 SKUMAST		
 Field Definitions		
	IMSKU	Packed(5,0) Part/Inventory Number
	IMNAME	Char(40) Part Name
	IMONHAND	Packed(7,0) Qty On Hand
	IMONORD	Packed(7,0) Qty On Order
	IMCOMMIT	Packed(7,0) Qty Committed
	IMUNITCOST	Packed(11,4) Unit Cost
	IMUNITPRIC	Packed(11,4) Unit Price
 Key Definitions		

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

Monarch changes the Data Structure from externally-defined to internally-defined, 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 in names as 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

Lastly, the field **NEWSKU** was added. If the positions of the new field were such that memory gaps would result, new fields would be created by Monarch (with enough length to fill the gap). See "Data Structures with Gaps" for more information.

---

## 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 by hand.

The Migrator will need to manually 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 the full context of its use.

---

## Other Code Generated by RPG Agent

### Printer O-Spec overflow and conditional EXCEPT support

Please refer to chapter 10 for detail description of code generated by RPG Agent to deal with print overflow and conditional EXCEPT support.

## Print related code (Net Print File defaults)

The Migration Directives defaults, and in particular the **“Net Print Files”** tab, allow the user to indicate additional print-related code to be generated by the RPG Agent.

### Insert **“/Error Please set printer name”** in Open Statements

Print files are explicitly opened to allow setup of print overrides, such as printer name, output directory, etc (see chapter 10).

Checking this default option will produce a line to be inserted in the migrated code, reminding the Migrator to set up a few properties **before** opening the file.

Remember that the /Error compiler directive is a way to stop the compilation and prompt the user to change a few lines of 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)
.
.
.
AFFMSTPP.Printer = "HP LaserJet"
Open AFFMSTPP DB(MonarchJob.PrinterDB)
EndConstructor

```

The following defaults are mutually-exclusive:

#### Use this printer name in migrated .Net Print files: XXXX

When this option is selected, the name of the printer indicated is burned-into the new DataGate Print File. This technique may be used when, during deployment, certain reports want to be sent to different printers for convenience. For example, orders-related reports may be desirable to print to the sales department printer, invoices to the accounting department, resumes to the human resources department printer, and so on.

#### Generate code to set the "Printer" property in migrated program, using: YY

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 opening it. Setting the property overrides any burned-in printer name the file may have.

#### Don't send output to printer, instead leave manuscript in pathname: PPP

It may be desirable to queue all reports into a specified directory on the server, so that other programs may process them later. PPP 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

**Examples:**

- Manuscript pathname: `C:\Production\UserManuscripts\Invoice.apm`  
 Result: Instead of sending the report to a printer, the file `C:\Production\UserManuscripts\Invoice.apm` is created (by default the .apm extension is associated with ASNA DataGate Renderer application, so double-clicking on this file will show a Print Preview on the server, and from there it can be sent to a printer).
- Manuscript pathname: `C:\Production\UserManuscripts\{0}\Invoice.apm`  
 Result: Assuming that the Printer file is named `QPRINT`, instead of sending the report to a printer, the file `C:\Production\UserManuscripts\QPRINT\Invoice.apm` is created (by default the .apm extension is associated with ASNA DataGate Renderer application, so double-clicking on this file will show a Print Preview on the server, and from there it can be sent to a printer).
- Manuscript pathname: `C:\Production\UserManuscripts\{1}\Invoice.apm`  
 Result: Assuming that the Job name is `ASP`, instead of sending the report to a printer, the file `C:\Production\UserManuscripts\ASP\Invoice.apm` is created (by default the .apm extension is associated with ASNA DataGate Renderer application, so double-clicking on this file will show a Print Preview on the server, and from there it can be sent to a printer).
- Manuscript pathname: `C:\Production\UserManuscripts\{1}\{2}\Invoice.apm`  
 Result: Assuming that the Job name is `ASP`, and the Job user is `PAUL` instead of sending the report to a printer, the file `C:\Production\UserManuscripts\ASP\PAUL\Invoice.apm` is created (by default the .apm extension is associated with ASNA DataGate Renderer application, so double-clicking on this file will show a Print Preview on the server, and from there it can be sent to a printer).
- Manuscript pathname:  
`C:\Production\UserManuscripts\{1}_{3}\{2}\Invoice.apm`  
 Result: Assuming that the Job name is `ASP`, and the Job user is `PAUL`, and the Job number is `1604`, instead of sending the report to a printer, the file `C:\Production\UserManuscripts\ASP_1604\PAUL\Invoice.apm` is created (by default the .apm extension is associated with ASNA DataGate Renderer application, so double-clicking on this file will show a Print Preview on the server, and from there it can be sent to a printer).

## Chapter 10 – 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, this type of O-Spec usage are referred by Monarch as "*Printer O-Specs*".

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

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

---

### Discovering 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 iSeries). The Printer O-Spec object type is basically 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, (the C-Spec with EXCEPT op-codes), does not define the Printer file, but rather it uses it as part of the program logic to trigger specified formats to be sent to the printer.

Given that the Monarch Printer O-Spec object type does not exist on the iSeries, but is a new Monarch concept, the collection/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/GamePlan is accessible.

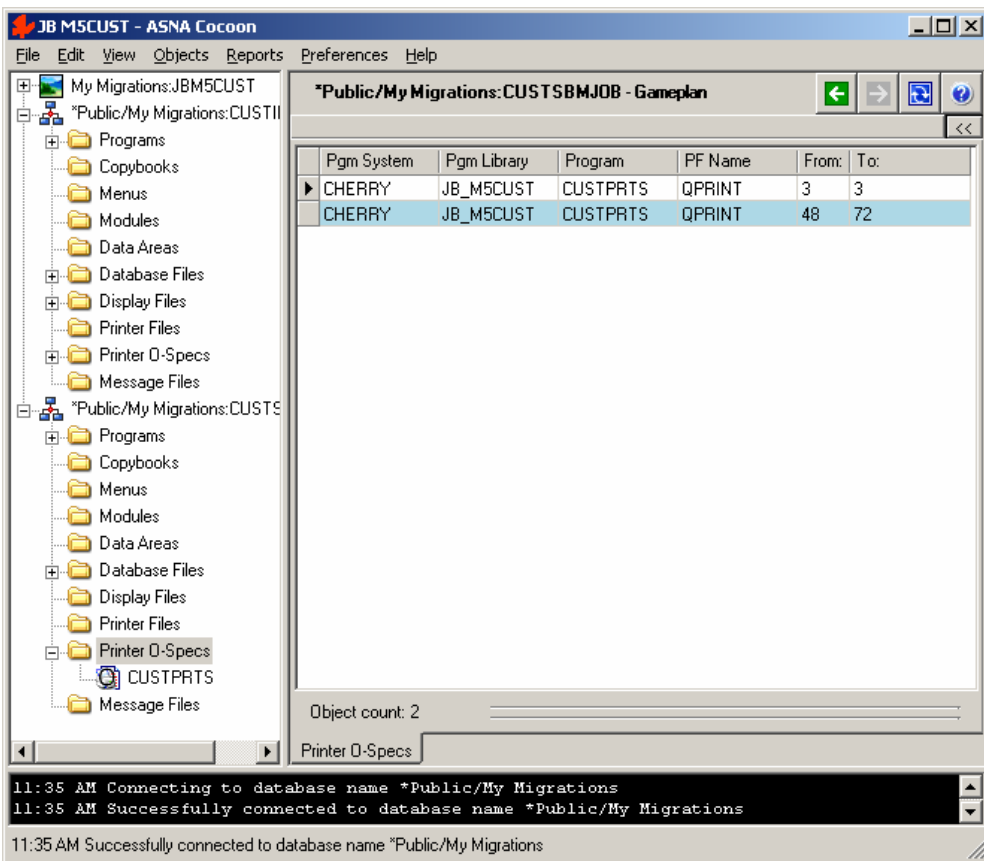
The Printer O-Spec object type directly depends on the legacy source of the program that describes and uses it. The dependency is so tight, that if the legacy source for the program changes, the Printer O-Spec needs to be re-discovered. The same way, if the program is removed from the Gallery/GamePlan the corresponding Printer O-Spec becomes orphan and unusable by Monarch.

To run the process to create Printer O-Spec objects, select a Gallery or GamePlan from Monarch Solution Explorer and execute "Discover Printer O-Specs" menu option, from the "Objects" Cocoon main menu.

After the Printer O-Spec discovery process completes, you should see new objects under the “Printer O-Specs” Gallery or GamePlan folder. The name of the objects is the same name as the program that defines it.

For every Printer O-Spec, the folder’s Grid view will list two entries. Each grid entry represents a section of program legacy source code where the Printer O-Spec is defined. The first section of code refers to the range of lines of code where the F-Spec is specified; the second section refers to the range of lines of code where O-Spec lines that define the O-Spec record format(s).

As you can see in the following figure, it shows the Grid View display for all Printer O-Specs on a GamePlan. Notice that there is only one Printer O-Spec named “CUSTPRTS” that was found in a program of the same name. It refers to lines 3 and 48-72 which are the two program’s legacy source code range of code lines that will be used when migrating the Printer file.



*Printer O-Spec grid entries*

The Legacy Source view for a Printer O-Spec displays the same program source code line-ranges as depicted by the Grid View, but in this case, the actual contents of the lines referred are displayed.

As can be seen in the figure below, the Printer "CUSTPRTS" is defined by a printer file named QPRINT that uses indicator \*INOF for page overflow logic and defines three record formats, namely: **PrtHeading**, **PrtDetail** and **PrtCount**.

```

Line Count: 26      Col      BROWSE      JB_MSCUST/QRPGSRC
                  :
FMT **  ...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8 ...+...
***** F Specification *****
0004.00  QPRINT  O  F 190      PRINTER OFLIND(*InOF)
***** O Specification *****
0049.00  QPRINT  E          PrtHeading  1 2
0050.00  O          'Sales Report For Customer:'
0051.00  O          CMNAME          +2
0052.00  O          'Printed'      114
0053.00  O          UPDATE          Y  +1
0054.00  O
0055.00  O  E          PrtDetail  1
0056.00  O          wPrtYr          Z  4
0057.00  O          CSSALES01       J  +0
0058.00  O          CSSALES02       J  +0
0059.00  O          CSSALES03       J  +0
0060.00  O          CSSALES04       J  +0
0061.00  O          CSSALES05       J  +0
0062.00  O          CSSALES06       J  +0
0063.00  O          CSSALES07       J  +0
0064.00  O          CSSALES08       J  +0
0065.00  O          CSSALES09       J  +0
0066.00  O          CSSALES10       J  +0
0067.00  O          CSSALES11       J  +0
0068.00  O          CSSALES12       J  +0
0069.00  O
0070.00  O  E          PrtCount  2
0071.00  O          wCount          3  18
0072.00  O          '+1 records printed for'
0073.00  O          CMNAME          +2
***** End of Specification *****
    
```

*Printer O-Spec object Legacy Source*

The layout of each format is described by the O-Specs in a textual format. Each format consist of one or more O-Spec record lines, and the lines in turn several O-Spec field lines. As you can see in Figure YYY, 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, known as the Printer O-Spec Data View.

The Printer O-Spec Data View tab, shown in the Figure below, describes the same printer file as the listing in the Figure above (*Printer O-Spec object Legacy Source*).

The Printer name is the root node of the tree ("QPRINT" in the figure), 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 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").

Name	Conditions	Directives	Comments
<b>QPRINT</b>			
<b>PRTHEADING</b>			1 Lines
<b>PRTHEADING(1)</b>		SpaceAfter(1) SkipBefore(2)	
'Sales Report For Customer.'			
CMNAME		EndPosition(+2)	
'Printed'		EndPosition(114)	
UPDATE		EndPosition(+1) DataFormat('Y')	
<b>PRTDETAIL</b>			1 Lines
<b>PRTDETAIL(1)</b>		SpaceBefore(1)	
WPRTYR		EndPosition(4) DataFormat('Z')	
CSSALES01		DataFormat('J')	
CSSALES02		DataFormat('J')	
CSSALES03		DataFormat('J')	
CSSALES04		DataFormat('J')	
CSSALES05		DataFormat('J')	
CSSALES06		DataFormat('J')	
CSSALES07		DataFormat('J')	
CSSALES08		DataFormat('J')	
CSSALES09		DataFormat('J')	
CSSALES10		DataFormat('J')	
CSSALES11		DataFormat('J')	
CSSALES12		DataFormat('J')	
<b>PRTCOUNT</b>			1 Lines
<b>PRTCOUNT(1)</b>		SpaceBefore(2)	
WPCOUNT		EndPosition(18) DataFormat('3')	
'records printed for'		EndPosition(+1)	
CMNAME		EndPosition(+2)	

Printer O-Spec object Data Format

Monarch shows each line of a record format as children 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, in Figure ZZZ, the record format “**PRTDETAIL**”, is composed of one line: **PRTDETAIL(1)**. If more line were defined for record format “**PRTDETAIL**”, then Monarch would list them as **PRTDETAIL(2)**, **PRTDETAIL(3)**, **PRTDETAIL(4)**, etc.

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

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

Also, by inspecting the information in Figure ZZZ, 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', such that the last character – the 'd' in this case – is printed on position 114.
5. One blank is printed at the end of the constant 'Printed', ready to print 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

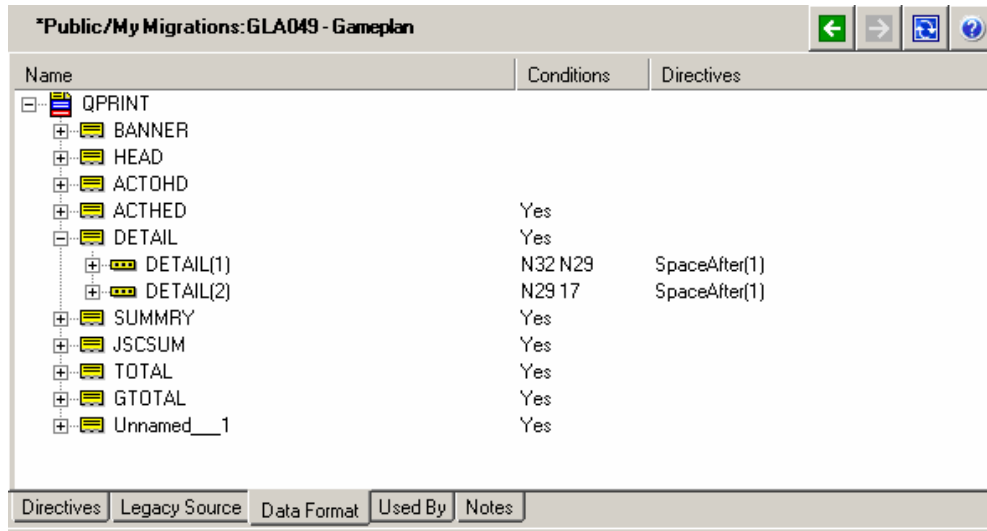
RPG allows naming exception records so that records with the same name can be grouped together. This name – referred by RPG as the EXCEPT name – is optional on iSeries RPG, but is required for Monarch to be able to create a DataGate Print file record format.

During the process of discovery, Monarch will create unique names (within the Printer O-Spec object). The naming convention used is: `Unnamed__{0}`, where {0} is a place holder for a unique auto-increasing number.

### Conditioned Field lines

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

For example, the Figure below shows the Data Format view for Printer O-Spec `QTEMP`, with ten record formats: `BANNER`, `HEAD`, `ACTOHD`, `ACTHED`, `DETAIL`, `SUMMARY`, `JSCSUM`, `TOTAL`, `GTOTAL` and an unnamed record.



Printer O-Spec record with conditioned lines

Notice how the **DETAIL** record has two lines that are conditioned, where the first line prints only when **\*IN32** is **\*OFF** and **\*IN29** is **\*OFF**; similarly, the second line prints only if **\*IN29** is **\*OFF** and **\*IN17** is **\*On**.

The corresponding legacy source is shown in the Figure below.

```

*Public/My Migrations: GLA049 - Gameplan
Line Count: 208 Col 21 BROWSE TRNGTSTFNS/QRPGLESRC
: GLA049
FMT ** ...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8 ...+...
0942.00 ** DETAIL LINE
0943.00JMRO10 E N29N32 DETAIL 1
0944.00JKHO10 XKXFFD 11 ' / & '
0945.00KDND1
0946.00KDND10 TBTCH 2 17
0947.00KDND10 TBAPPL 20
0948.00KDND10 TBCODE 23
0949.00KDND10 TBREF# 36
0950.00KDND10 TBJRNL 49
0951.00KDND10 WDCHCK 58
0952.00 *
0953.00KDND10 TBDESC 84
0954.00 *
0955.00KDND10 N21 TBTAMT 3 98
0956.00KDND10 21 TBTAMT 3 111
0957.00KDND10 RUNBAL 1 126
0958.00KDND10 26 128 'CR'
0959.00KDND10 UNPOST 130
0960.00 *
0961.00GLED10 E N29 17 DETAIL 1
0962.00KDND10 WDWNER 71
  
```

Legacy source for QTEMP Printer O-Spec

Monarch's Printer O-Spec Agent will split record **DETAIL** into two:

```

DETAIL1_IF_N32_N29 and
DETAIL2_IF_N29_17
  
```

Remember that the migration of the O Specifications into a DataGate Print File happens before the application is compiled, way 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 like the following:

```
ExSr Write_DETAIL
```

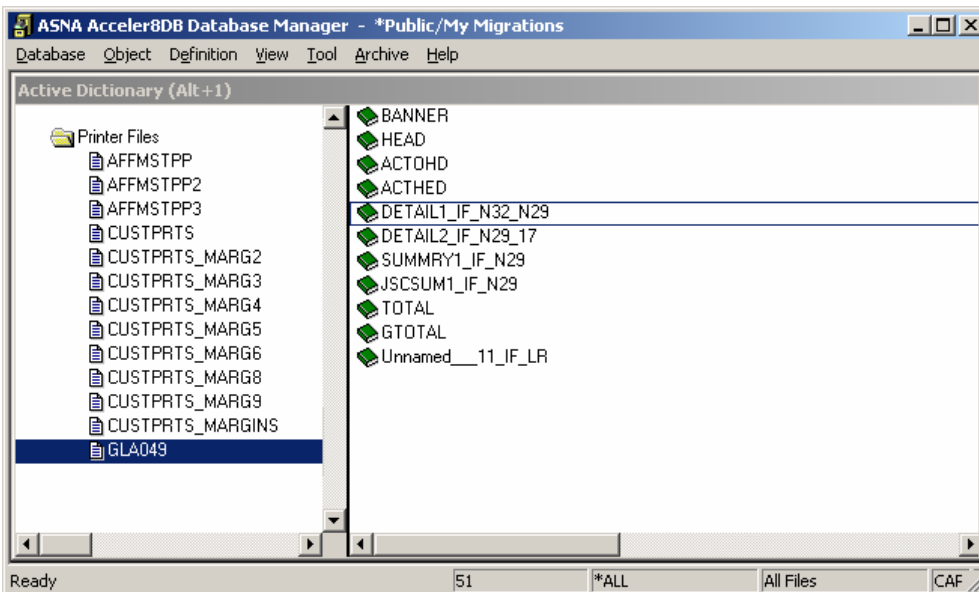
Where:

```

BegSr Write_DETAIL
  If ( *IN32 = *Off *And *IN29 = *Off )
    Write DETAIL1_IF_N32_N29
  EndIf
  If ( *IN29 = *Off *And *IN17 = *On )
    Write DETAIL2_IF_N29_17
  EndIf
  
```

EndSr

Effectively preserving the application logic, where the indicators may be tested when the application executes.



Migrated *QTEMP* DataGate Print file

## Migration Directives

Print files on the iSeries 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 - the measurement CPI was used.

The CPI unit represents a width measurement by counting the number of characters that can be printed in one inch of paper.

Typical CPIs used were:

1. 10 CPI, *normal* font width to print 80 characters on a letter paper (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 6 lines per inch, a total of 66 lines per page on a 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.

Directives such as `spaceBefore(3)`, means: leave three fixed-height printers row blank before printing this record format; `'DISTRIBUTION SUMMARY' EndPosition(72)` means: print constant in this line in such position so that the last character of the label is printed in position 72.

Monarch Migrated applications run on .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 raster devices that can print in almost any position in the paper. Typical printer resolutions are measured in DPI horizontally and vertically.

The DPI unit represents the amount of *Dots* per Inch. 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 a character/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 be printed at position 14, line 5 using 10 CPI fonts, would be printed at:

$$14 \times ( 1 / 10 ) = 1.4 \text{ inches horizontally}$$

$$5 \times ( 1 / 6 ) = 0.83 \text{ inches vertically}$$

During the Printer O-Spec Migration, Monarch needs to convert positions and lines to inches (or centimeters). Instead of blindly using 0.1 and 0.16 as the factors to multiply positions and lines by to obtain the English or metric units, Monarch allows the migrator to specify conversion factors.

Especially important during the modernization of a report where the Migrator might want to use a variable-pitch Font, the ability to adjust the conversion factor becomes even more crucial.

The factor that converts positions to horizontal inches is called - *in Monarch terminology*-, "Average char width".

The factor that converts line counts to vertical inches is called - *in Monarch terminology* -, "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 an 'm' would be the widest letter and the 'i' the thinnest.

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

The figure TTT shows the Printer O-Spec directives tab. On the left hand side, under the "Legacy design" group, Monarch shows the printer metrics corresponding to the QPRINT default printer file on the iSeries. On the right-hand side, under the "Migration parameters" Monarch presents the parameters that will be used during the conversion.

## Margins Group

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 it will vary from printer to printer how much text will be missing if attempting to print too close to the right edge of the page.

A margin value of **\*None** is used to indicate that the left-most position of the layout should print as close to the left of the page as the printer possibly can.

## Font Group

The conversion factors mentioned above, that is, the "Average char width" and the "Line Height" are the result of selecting a .NET Font with a specific size. As fonts are selected and/or sizes are entered, Monarch computes the average width and height of the font, as closest as possible to the width of a character printed using the default printer installed in the Migrator's computer.

The resulting Average char width and Line Height may be overridden by entering new values. As long as the cursor is not positioned on the Name and/or Size, the values of the factors will not be changed by Monarch.

The screenshot shows the 'Printer O-Spec Directives' dialog box in the Monarch software. The dialog is titled '\*Public/My Migrations: CUSTPRTS - Gameplan'. It is divided into several sections:

- Origin:** Gallery: My Migrations:JBM5CUST, Library List: JB\_M5CUST
- New Object Location:** Database name: \*Public/My Migrations, Library: PRINTER FILES, .Net printfile name: CUSTPRTS
- Legacy design (\*):** Page Length: 66, Page Width: 132, Characters per Inch: 10, Lines per Inch: 6, Overflow line: 66. A note below indicates '\* Default QPRINT design'.
- Migration parameters:** Includes a checked 'Migrate' checkbox and an unchecked 'Use default directives' checkbox.
- Margins:** Left: 0.75000000, Top: \*None, Inches.
- Font:** Name: Courier New (with a 'Select...' button), Point size: 8.25, Average char width: 0.06871049, Line Height: 0.15183757, Inches.
- Paper Kind:** Legal (dropdown), Width: 8.5, Height: 14, Inches.
- Orientation:** Portrait (selected), Landscape.
- Overflow:** 0.00000000, Inches. A 'Restore Defaults' button is present.

At the bottom, there are tabs for 'Directives', 'Legacy Source', 'Data Format', 'Used By', and 'Notes'.

*Printer O-Spec Directives*

## Paper Kind Group

Select any of the listed .NET Paper kinds. Monarch will show the physical width and height of the corresponding paper for reference. If none of the Paper kind dimensions is 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 of the fields and constants that make up the report.

### Orientation Group

Select the orientation of the characters to be printed. When using Landscape, the print head will start print as if the paper were rotated 90 degrees CCW.

### Overflow

When fetching for overflow - see "Handling Page 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: the overflow area starts when the printer head is 1.5 inches away from the end of the page.

### Restore Defaults Button

Pushing this button will copy the *current* Net Print file Migration Directive defaults (which may have changed since the Printer O-Specs were discovered). Notice how there is no way to change the units of measure from this screen; when restoring the defaults the new values will be copied along with their proper unit of measurement.

## 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 record **WRITESs**. Additionally, whenever an EXCEPT line gets 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
```

Where:

```
BegSr Write_DETAIL
  If ( *IN32 = *Off *And *IN29 = *Off )
    Write DETAIL1_IF_N32_N29
  EndIf
  If ( *IN29 = *Off *And *IN17 = *On )
```

```

Write DETAIL2_IF_N29_17
EndIf
EndSr

```

## 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 a special indicator for such purpose (although any of the numbered indicator may also be used), the indicator `oF`. ASNA AVR *does not* recognize OF as \*INOF, but allows to declare fields of type indicator, that is `Type(*Ind)`.

Additionally, ASNA's AVR new 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 the legacy page overflow handling.

For example, the following line is generated by Monarch's RPG Agent for a program that prints using O-Specs and has overflow handling logic:

```

DclFld IndicatorOF Type(*Ind)

DclPrintFile QPRINT DB(MyJob.MyPrinterDB) File("PAY080") ImpOpen(*No)
OverflowInd(IndicatorOF)

```

It first declares a field called `IndicatorOF` and then passes the field to the declaration of the Print file, using the keyword `overflowInd()`.

Notice how the name `IndicatorOF` was chosen with a name longer than ten characters, to avoid clashing with any other field declared in RPG legacy code. Also notice that \*INOF cannot be used, since it is not part of AVR grammar and \*IN prefix is reserved for numeric indicators.

### 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.

Name	Conditions	Directives
QPRINT		
BANNER		
HEADNG	Yes	
HEADNG(1)	OF	SpaceAfter(1) SkipBefore(01)
HEADNG(2)	OF	SpaceAfter(1)
HEADNG(3)	OF	SpaceAfter(2)
HEADNG(4)	OF	SpaceAfter(1)
HEADNG(5)	OF	SpaceAfter(1)
HEADNG(6)	OF	SpaceAfter(1)
HEADNG(7)	OF	SpaceAfter(2)
REPORT	Yes	
REPORT(1)		FetchOverflow(*True) SpaceAfter(1)
EMPTOT		
TOTLIN		

*Data Format view of a printer file that uses Overflow detection*

The Printer O-Spec in the Figure above shows printer file `QPRINT` with five record formats: `BANNER`, `HEADNG`, `REPORT`, `EMPTOTAL` and `TOTLIN`. Notice how `HEADING` lines one thru seven are "*conditioned*" on Overflow. Also notice how record format `REPORT` includes the directive `FetchOverflow(*True)`.

The logic reflects the fact that when `REPORT` gets written, the overflow indicator needs to be checked, and if set, all the `HEADNG` lines need to be printed *before* the `REPORT` lines get also printed.

Perhaps it is easier to understand the logic by looking at the Monarch generated code. The `EXCEPT REPORT`, gets migrated as:

```

        ExSr Write_REPORT
. . .
        TOTAL = TOTAL + 1
        TYPTOT = TYPTOT + 1

```

where, `Write_REPORT` is a method implemented as:

```

BegSr Write_REPORT
    ExSr QPRINTOverflow
    Write REPORT
EndSr

```

and Monarch generates the implementation of `QPRINTOverflow` as:

```

BegSr QPRINTOverflow
    If IndicatorOF = *On
        Write HEADNG
    EndIf
EndSr

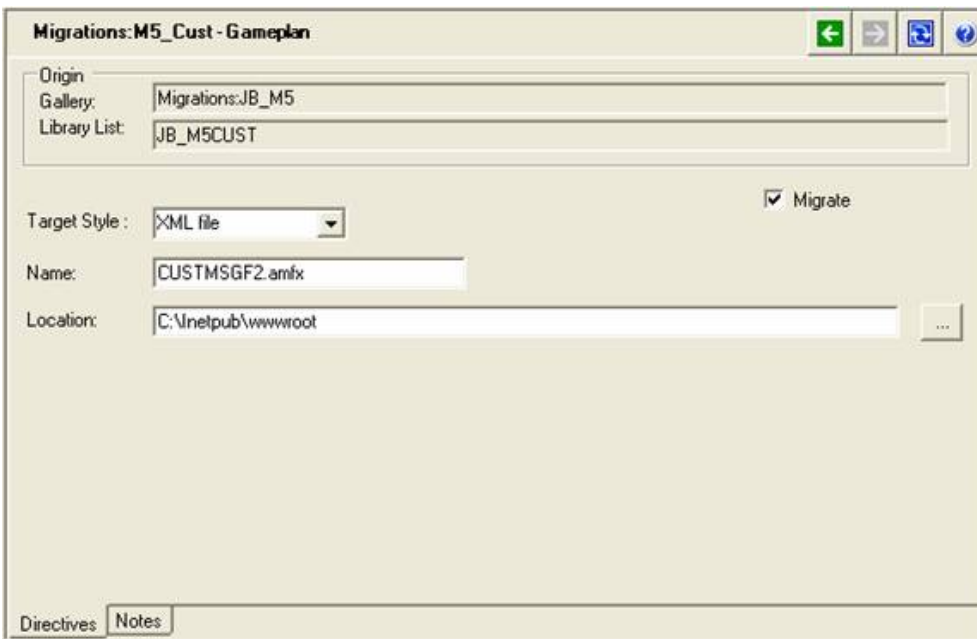
```

The result is that every time the `ExSr Write_REPORT` gets executed, the value of field `IndicatorOF` is checked (the value of `IndicatorOF` is changed by the AVR runtime *after* writing to *any* DataGate print file format). If the `IndicatorOF` is `*On`, meaning that the *next* line *would* print inside the page overflow area (as defined by the Printer-O Spec object by the negative overflow value in the directives). The application has the chance to Skip to a line in the next page, print heading lines and continue printing the rest of the report.

This page intentionally left blank

## Chapter 11 – Migrating Message Files

The following figure shows the Directives page for Message Files.



A migrated message file becomes an **XML document**. The following figure shows an example:

```
<!-- Message File JB_M5CUST/CUSTMSGF Migrated on 6/22/2005 at 11:40 AM by
  ASNA Monarch(R) version 2.0.284.0 -->
<LegacyMsgFileContents version="1.0">
<messages>
  <MSG MSGID="CST0001" SEVERITY="0" DATACOUNT="1" TEXT="Customer &1
    has been added" SECLVL="" TYPE1="*CHAR" LEN1="9" DEC1="0" />
  <MSG MSGID="CST0004" SEVERITY="20" DATACOUNT="0" TEXT="Prompting
    is available for State and Status only." SECLVL="" />
  <MSG MSGID="CST0005" SEVERITY="0" DATACOUNT="1" TEXT="&1
    customer(s) were submitted to batch for processing." SECLVL=""
    TYPE1="*DEC" LEN1="3" DEC1="0" />
  <MSG MSGID="CST0006" SEVERITY="0" DATACOUNT="1" TEXT="The sales
    report has been printed." SECLVL="" TYPE1="*DEC" LEN1="7" DEC1="0" />
  <MSG MSGID="CST1001" SEVERITY="20" DATACOUNT="1" TEXT="The &1
    cannot be blank." SECLVL="" TYPE1="*CHAR" LEN1="25" DEC1="0" />
```

```

<MSG MSGID="CST1015" SEVERITY="20" DATACOUNT="3" TEXT="Library &1
is not accessible." SECLVL="The system returned an error &2 when trying
to execute &3 on library &1." TYPE1="*CHAR" LEN1="10" DEC1="0"
TYPE2="*CHAR" LEN2="7" DEC2="0" TYPE3="*CHAR" LEN3="20" DEC3="0"
/>
</messages>
</LegacyMsgFileContents>

```

## Install Directory

When an application using a message file runs, it will look for the file following this algorithm:

1. If the WebJob.CurrentJob.MessageFileFolder is not null, its contents are considered an absolute path to the folder containing message files. <end>
2. The Web.config file in the application's folder is searched for an application setting of the form:

```

<configuration>
  <appSettings>
    <add key="MonarchMessageFileFolder"
      value="C:\Inetpub\wwwroot\BenMaint\" />
  </appSettings>
  <system.web>
    ...
  </configuration>

```

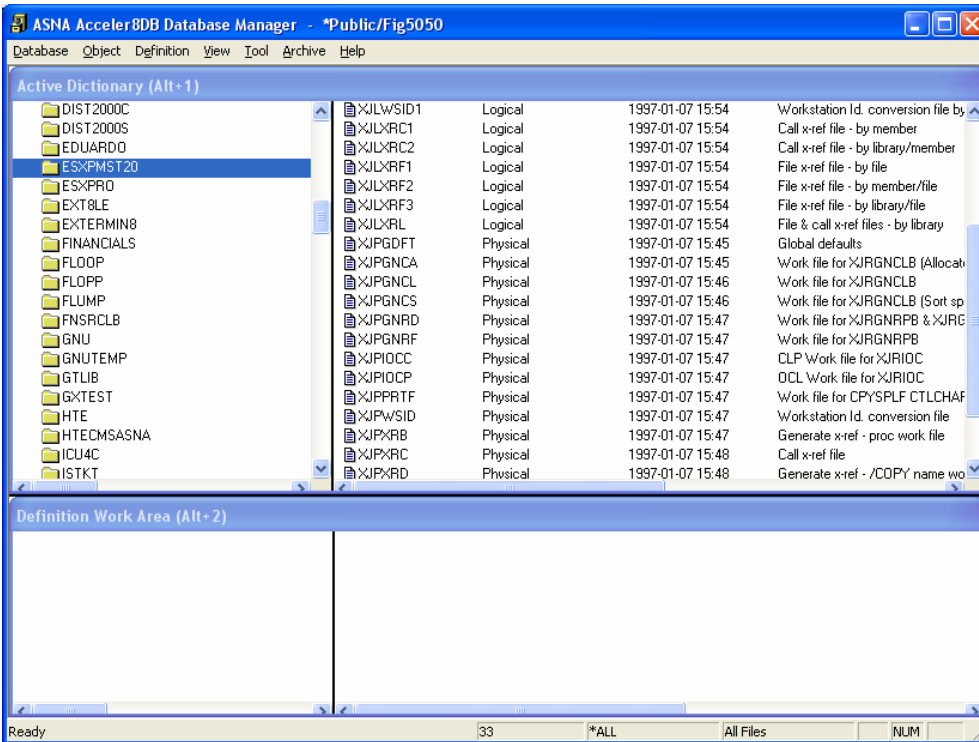
If it is found, the value entry is considered the folder containing message files. <end>

3. The arbitrary folder C:\Inetpub\wwwroot is considered the folder containing message files. <end>

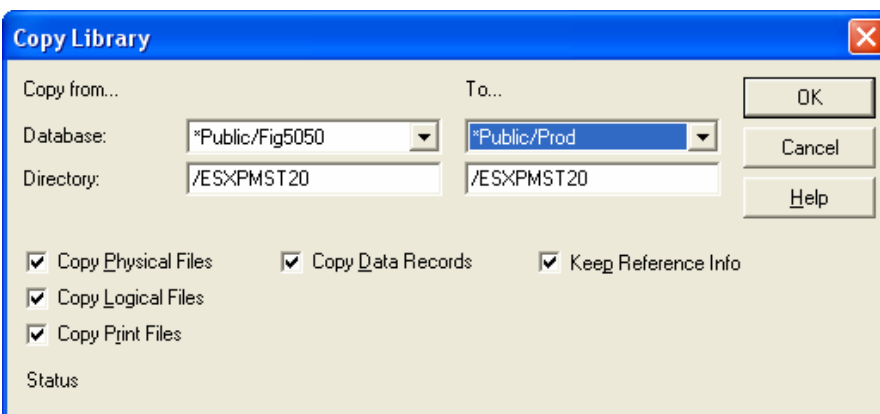
## Chapter 12 – Migrating Data

Data can remain on the iSeries or it can be moved to a new database platform like Microsoft SQL Server.

### Using Database Manager



### Copying a Complete Library



## Copying a Single Physical File

**Copy Data**

Copy from... To...

Database: \*Public/Fig5050 \*Public/Fig5050

File: /ESXPMST20/XJPGDFT /ESXPMST20/XJPGDFT

Member: \*FIRST \*SAME

Browse... Browse...

Field Mapping:  
 No Map - Format IDs must be identical  Keep Reference Info

Copy mode

Number of records to copy: All Records

Copy in BRN order  Copy in Key order

Copy records in range...

First BRN:

Last BRN:

OK  
Cancel  
Help

## Copying a Single Logical File

**Copy Logical File**

Copy from... To...

Database: \*Public/Fig5050 \*Public/Fig5050

Scope library: /ESXPMST20 /ESXPMST20

File sub-path: XJLXRC2

Member: \*FIRST

File: \*SAME

Copy base file definitions

Replace target logical file

Replace target base files

OK  
Cancel  
Help

## Considerations for MS SQL Server

If data files are to be moved to SQL Server to be used at run-time via DataGate for SQL Server (DSS), then DSS restrictions apply to the migrated application.

Even though DSS tries to make *SQL Server* look like DB2/400, there are several features that can't be implemented in a totally transparent fashion.

In the next sections we'll deal with all the issues that will need your attention, but the following items are probably the ones with the most impact for many of you.

### No multi-format logical files

DSS implements a physical file through the use of a native *SQL Server* table. A logical file is implemented through a native view. Views are single-formatted in nature, so there is no support for multi-format logical data files. You will have to eliminate any reference to multi-format logical files in your application.

Print files, which are typically multi-formatted, are fully supported in DSS.

### Single member files

*SQL Server* doesn't have the concept of members of a table/view. DSS makes it appear as though each file had one (and only one) member; as the member name is exactly the same as the file name. For the member name, you can use the exact name or the special values \*FIRST and \*FILE. You should be careful when using the Copy Data and Copy Library tools of Database Manager to copy a file from ADB or iSeries, because they default to \*SAME for the target member name. If the source member name was differently than the file name, the copy will fail.

If your application depends upon the existence of multiple members per file, you will have to re-architect it to provide an alternative method for handling the logic involving these files.

### Logical field restrictions

There are two restrictions on the usage of logical fields when they change the name or the type of their corresponding base physical field. When the field is retyped (most typically because the field is a concatenation or substring of the physical) then the field becomes read-only. A logical field, whose name has changed from its physical base field can't be used as a key field in the logical file.

### No QTemp

Version 7 of DSS does not support the special Library QTEMP.

### Unlocking records

This is probably the most demanding area of application adaptation. The problem arises in two areas.

1. Using the Unlock keyword on the read operations.
2. The implementation of the operation Unlock.

DSS uses *SQL Server* 'Server Cursors' to implement file access. When a file is opened for update, it is not possible to tell *SQL Server* not to lock the record on a read, so this option is not valid for files opened for Update. You have two options to solve this problem.

1. Declare a second instance of the file marked as input only and use it wherever the NoLock option was given on a **Read/Chain**.
2. Leave the NoLock and follow the **Read/Chain** with an unlock; however this option has the problems stated in the next paragraph.

The other problem is in the use of the Unlock operation. **Unlock** leaves the cursor in a no-position state, meaning you can't perform a subsequent read(next/previous) without repositioning the file with a **Set(LL/GT)** or **Chain**.

## 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 iSeries allows GOTO from subroutine code to mainline code. Branching explicitly from one member into 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           .
C*           Go to TAG2 on some condition
C           GOTO  TAG2
C           .
C           .
C           ENDSR
C*           SUBRC    BEGSR
C           .
C           .
C*           Go to TAG3 on some condition
C           GOTO  TAG3
C           .
C           .
C*           Go to TAG1 on some other condition
C           GOTO  TAG1
C           .
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 )
  ENDRY
// 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
```

## How to Migrate Embedded SQL

Use ADO.NET to migrate imbedded SQL.

The following figure shows a typical usage of embedded SQL.

```
C*-- Prepare and Open Cursor -----
*
C/EXEC SQL
C+ PREPARE STMT1 FROM :STMT1
C/END-EXEC
*
C/EXEC SQL
C+ DECLARE C1 CURSOR FOR STMT1
C/END-EXEC
*
C/EXEC SQL OPEN C1
C/END-EXEC

C*-- Read selected data and populate Subfile -----
C          Z-ADD      1          RECNO
C/EXEC SQL
C+ FETCH C1 INTO  :DSINPUT
C/END-EXEC
C          SQLCOD      DOWEQ      0
C          EXSR        LOADSF
C          ADD         1          RECNO
C/EXEC SQL
C+ FETCH C1 INTO  :DSINPUT
C/END-EXEC
C          ENDDO
*
C*-- Close Cursor-----
*
C/EXEC SQL CLOSE C1
C/END-EXEC
```

Here is a possible translation into AVR using ADO.

```
//-----
// Note that this method uses the Client Access ODBC driver.
//-----
DclFld SQL_Adapter Type(System.Data.Odbc.OdbcDataAdapter)
DclFld SQL_Cmd Type(System.Data.Odbc.OdbcCommand)
DclFld SQL_Dataset Type(System.Data.DataSet) New()
DclFld SQL_Connection Type(System.Data.Odbc.OdbcConnection)
DclFld SQL_Table Type(System.Data.DataTable)

// Establish a connection to the server
```

```
SQL_Connection = *New OdbcConnection("Driver={Client Access ODBC Driver+
    (32-bit)};System=Production;Uid=MyUser;Pwd=MyPassword")
SQL_Connection.Open()

// Prepare and open Dataset
SQL_Cmd = *New OdbcCommand(Stmt1, SQL_Connection)
SQL_Adapter = *New System.Data.Odbc.OdbcDataAdapter(SQL_Cmd)
Open MemoryFile
SQL_Adapter.Fill(SQL_DataSet)
SQL_DataSet.Tables[0].TableName = "RCINVT"
MemoryFile.DataSet.Merge(SQL_DataSet.Tables[0])

// Read selected data and populate subfile
RECNO = 1
Chain SQL_Cursor Key(RECNO) NotFnd(SQLCOD)
DoWhile (SQLCOD = 0)
    EXSR LOADSF
    RECNO = RECNO + 1
    Chain SQL_Cursor Key(RECNO) NotFnd(SQLCOD)
EndDo

// Close MemoryFile and Connection
Close MemoryFile
SQL_Connection.Close()
```

This Page Intentionally Left Blank.

## Chapter 14 – Deploying the New Application

### First the basics.

Before we drill into how to deploy a Web application, let's first cover a few basics. Monarch applications are a special kind of AVR for .NET Web application that requires the following additional components:

1. Monarch Web Controls. (ASNA.Monarch.WebDspF.dll)
2. Monarch Framework. (included in ASNA.VisualRPG.Runtime.dll)
3. Monarch Java Scripts. (Common.js and WinPopUp.js)
4. Your application Message files.

Installing **DataGate WebPak** on the server, takes care of installing ASNA.VisualRPG.Runtime.dll, ASNA.Monarch.WebDspF.dll assemblies and the Monarch Java Script support. Your application Message files need to be copied manually (see discussion below).

Be certain to properly license ASNA DataGate *WebPak*.

If you experience problems with your application, such as function keys not responding, verify that DataGate *WebPak* installed the Java Script support installed on your server, inside the `aspnet_client` folder:

```

\\MyServer\MyDefaultWebSite\aspnet_client\Monarch\2_0\
                                     Common.js
                                     WinPopUp.js

```

### Installing Application Message files

If your application uses message files, the migrated XML files need to be copied to the server. Look in the application's Web.config file to get the path where the application will look for Message files.

```

<appSettings>
  <!-- MIGRATED APPLICATION PROPERTY SETTINGS GO HERE.
        Example: <add key="settingName" value="settingValue" />
  -->
  <add key="MonarchMessageFileFolder" value="C:\Monarch MsgFiles\" />
</appSettings>

```

Identify the section "`appSettings`" and look for the value of the entry `MonarchMessageFileFolder`. Copy your message files from the development machine to the folder configured in Web.config.

## Monarch application deployment is no different from any AVR Web application

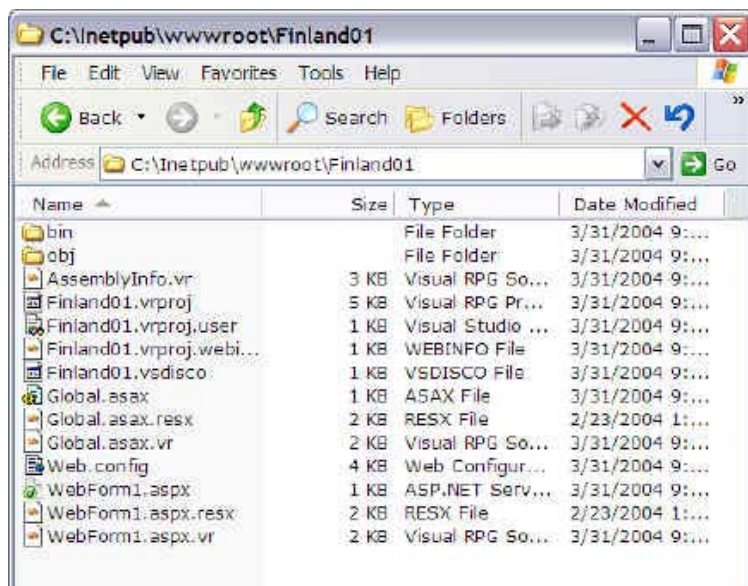
The following describes the considerations to deploy any AVR for .NET Web application, which applies to Monarch Applications as well.

AVR for .NET Web applications are built using the Microsoft .NET ASP.NET model. An ASP.NET application lives in a virtual directory on your Web server. A virtual directory ties a URL to a physical path on your Web server (one definition of a virtual directory is that it is a logical name you associate with a physical path).

If you use the manual method of deploying your Web application, you'll need to create your application's virtual directory yourself. If you use the Visual Studio Web setup project to install your Web application, the setup project will create the virtual directory for you.

### Folder Structure

Regardless of how you deploy your Web application, the folder structure on the Web server remains the same. The figure below shows the typical folder structure for an AVR for .NET Web app as it exists on a developer PC. The application source (and compiler-related files) resides in the **Finland01** folder; the executable assembly (the binary output) resides in the **Bin** folder under **Finland01**; and other compiler related files reside in the **obj** folder under **Finland01**.



Typical AVR for .NET Web app folder structure on developer PC

For development purposes, all of the **Finland01** folder contents (and its subfolders) are necessary for compiling the project. **Note:** *If you were going to give this project to another developer, you'd want to give them the entire contents of Finland01.* To deploy the app, though, you do not need all the files and folders you see here.

At the minimum, the Web server will need a top-level application folder and under it a **Bin** folder. For example, let's say you wanted to deploy the Finland01 app to your Web server under the `c:\webapps\` folder. In this case, your Web server folder structure might look like this:

```
c:\webapps\  
  
    finland01\  
  
        bin\  
  
        
```

The `\finland01\` folder would contain the application's Web pages (the ASPX pages and others--more on this in a moment) and the `\bin\` folder would contain the application's DLL(s). The virtual directory for this application would then be created over `\c:\webapps\finland01\`.

Manually deploying an AVR for .NET Web application is very simple. Beyond being a good way to deploy an app, the basic techniques outlined here are also used to refresh an app install after having made changes to that app. No matter how you ultimately deploy a Web app, understanding the steps to manually deploy a Web app are very important.

**Manually deploying an AVR for .NET Web app requires these basic steps for deploying any one application:**

1. Create an application top-level folder on the server.
2. Create a Bin folder immediately under the application's top-level folder.
3. Copy ASPX/ASCX files and Web.Config to the application's top-level folder.
4. Copy the application's DLL to the Bin folder.
5. Create a virtual directory over the application's top-level folder.

The image below shows a picture of the process outlined above. In this picture:

- The Global.ASAX is copied to the target root.
- The Web.Config is copied to the target root.
- All ASPX (and ASCX if present) files are copied to the target root.
- The project's DLL is copied from its Bin folder to a Bin folder within the target root.

---

ASP.NET project files deployed

## Additional server components needed

In addition to installing your project-specific files, you'll also need to install a few other files on your server.

You'll also need to have installed on the Web server, for *all* Web apps:

1. **The AVR for .NET Web deployment setup.** This installs the AVR for .NET framework needed by your applications at runtime, as well as the ASNA licensing component. This step only needs to be performed for initially installing the AVR for .NET on your Web server or when you need to update your Web server with a new version of AVR for .NET.

2. **The .NET Framework.** This is generally installed simply as a matter of course when you perform the Windows update, but may be installed manually. Regular use of Microsoft's Windows update facility will keep the .NET Framework current on your Web server.

## Deploying your project - A little more detail

At a minimum, in the application's top-level folder you need to deploy all the application's .ASPX files, its .ASCX files (if it used any user controls) and its Web.Config file. There may be other files, such as images, that you also need to deploy--based on what else you used to create the application. For most basic to intermediate applications, though, the files listed in the table below are the only ones you need to worry about.

File name	Purpose
*.ASPX	Web pages
*.ASCX	User control Web pages
Global.ASAX	Global ASAX file
Web.Config	Web configuration file

In the top-level folder's Bin folder, you need to deploy the application's DLLs and any other necessary DLLs (for example, if the application used a business not stored in the GAC, you'd deploy it to this Bin folder).

To deploy an application manually, after creating the top level folder, copy all the .ASPX, .ASCX (if any) and the Web.Config to the application's top level folder. Copy the application's DLL (and any subordinate DLLs) to the application's Bin folder. To copy these files you only need copy them from the development PC to the server. There is *no* registration of anything necessary.

For a simple three page app with two user controls and no subordinate DLLs, the folder structure would look like this:

```
c:\webapps\  
    finland01\  
        mainpage.aspx  
        secondpage.aspx  
        thirdpage.aspx  
        mycontrol1.ascx  
        mycontrol2.ascx  
        web.config  
        bin\  
            findland01.dll
```

To improve this process, consider using a DOS batch file. In the example below, the batch file deploys the entire Web application (rather than needing to determine what parts of the app have changed). With a tested batch file, deploying updated versions of the Web application is a snap.

```
xcopy *.aspx      \\myServer\MyAppFolder    /y
xcopy *.ascx      \\myServer\MyAppFolder    /y
xcopy web.config  \\myServer\MyAppFolder    /y
xcopy global.asax \\myServer\MyAppFolder    /y
xcopy bin\*.dll   \\myServer\MyAppFolder\bin /y
```

## Using Visual Studio's Web Setup Project to Deploy an AVR for .NET Web application

The output from a Visual Studio Web Setup Project is an **MSI** or **EXE** file that offers a one-click installation of your project. This file comprises the distribution for your application and when it is run, it installs all the application files to their appropriate target folders. This installation file additionally performs other installation tasks, such as creating target folders and creating the necessary virtual directory for you.

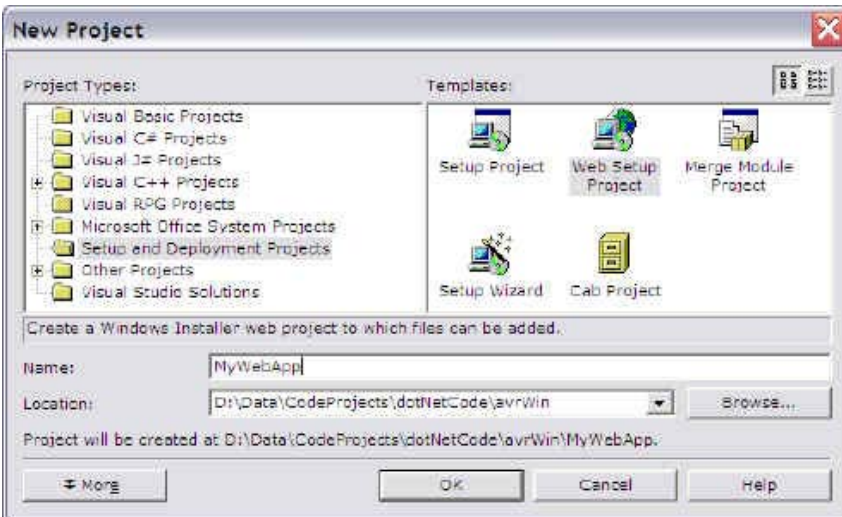
There are several options you can choose for exactly what the binary output is from the MS Installer. This article will focus on the **MSI** file format.

**Note:** To create a Visual Studio Web Setup Project, you must be using Visual Studio Professional version or higher.

### Creating an MSI Installation File

Creating the MSI installation file is a very easy. The MSI is created as a result of building a special-case Visual Studio project. When you build this project, its output is the binary installation file.

1. To create a Web Setup Project, open Visual Studio and select **File | New Project**.
2. Select the **Setup and Deployment Project** from the *Project Type* list on the left.
3. On the right, select **Web Setup Project** from the project *Templates* list.
4. Enter a **name** for the project in the appropriate text box. (Take care with this name, as it will be used as the default name for the virtual directory this setup project will create).



Ready to define a Web Setup Project

## Special considerations when deploying on Windows 2003 Server

(To be documented)

## Printers Available for Monarch Applications

AVR for .NET Web applications by default run under a restricted account with no printer installed.

In order for the application to print, it is necessary to change the user account under which ASP.NET is running to one that has sufficient permissions and contains printers installed (local or with sufficient permissions to network printers).

By default, ASP.NET runs with the permissions of the local “machine” (ASPNET account for the ASPNET worker process) account.

Let’s assume that there is an account named **BOB**, with printers installed.

**The following describes how to make the Framework to run under the local BOB account.**

1. To make this change, it is necessary to edit the default configuration settings in the Machine.config file.

This file is located by default at:

```
C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322\CONFIG\machine.config
```

The account setting is contained within the <ProcessModel> tag of this file.

The default setting for the account is:

```
userName="machine" password="AutoGenerate"
```

This can be changed to:

```
userName="BOB" password="BobsPassword"
```

**For example:**

```
<processModel enable="true" timeout="Infinite"
idleTimeout="Infinite" shutdownTimeout="0:00:05"
requestLimit="Infinite" requestQueueLimit="5000"
restartQueueLimit="10" memoryLimit="60" webGarden="false"
cpuMask="0xffffffff" userName="machine" password="AutoGenerate"
logLevel="Errors" clientConnectedCheck="0:00:05"
comAuthenticationLevel="Connect"
comImpersonationLevel="Impersonate"
responseDeadlockInterval="00:03:00" maxWorkerThreads="20"
maxIoThreads="20"/>

<processModel enable="true" timeout="Infinite"
idleTimeout="Infinite" shutdownTimeout="0:00:05"
requestLimit="Infinite" requestQueueLimit="5000"
restartQueueLimit="10" memoryLimit="60" webGarden="false"
cpuMask="0xffffffff" userName="BOB" password="BobsPassword"
logLevel="Errors" clientConnectedCheck="0:00:05"
comAuthenticationLevel="Connect"
comImpersonationLevel="Impersonate"
responseDeadlockInterval="00:03:00" maxWorkerThreads="20"
maxIoThreads="20"/>
```

2. Then save the file.

*The Server may need to be re-booted for this change to take effect*

## Glossary

**Cocoon** - A windows application, delivered as part of Monarch, used by a Migrator to perform his everyday tasks.

**Gallery** - A repository of descriptions for objects in one or more libraries of an iSeries used to analyze their relations and usages.

**GamePlan** - A database containing a plan collecting the strategies chosen to migrate a Subsystem of an application.

**Migrator** - A Programmer familiar with an application executing the migration with the help of Monarch.

**Migration Agent** - A specialized Program used in the migration of an OS/400 object type.

**Density** - A unit of complexity that measures the effort required in migrating a Program.