



*RPG IV Programming  
Intermediate Workshop for i*

(Course code AS07)

**Student Notebook**

ERC 5.0

Authorized

**IBM | Training**

## Trademarks

IBM® is a registered trademark of International Business Machines Corporation.

The following are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

AS/400®	Balance®	DB2®
i5/OS®	Integrated Language Environment®	iSeries®
Language Environment®	Notes®	Operating System/400®
OS/400®	Rational®	Redbooks®
RPG/400®	System i®	WebSphere®
400®		

Adobe is either a registered trademark or a trademark of Adobe Systems Incorporated in the United States, and/or other countries.

Pentium is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Windows is a trademark of Microsoft Corporation in the United States, other countries, or both.

UNIX® is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

### February 2009 edition

The information contained in this document has not been submitted to any formal IBM test and is distributed on an "as is" basis without any warranty either express or implied. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will result elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

# Contents

<b>Trademarks .....</b>	<b>xi</b>
<b>Course Description .....</b>	<b>xiii</b>
<b>Agenda .....</b>	<b>xvii</b>
<b>Unit 1. Welcome and administration.....</b>	<b>1-1</b>
Administration .....	1-2
Course objectives .....	1-3
Prerequisites .....	1-4
Agenda (1 of 2) .....	1-5
Agenda (2 of 2) .....	1-6
Introductions .....	1-7
<b>Unit 2. Managing display record formats.....</b>	<b>2-1</b>
Unit objectives .....	2-2
Inquiry example: Display file DDS .....	2-3
Inquiry example: Item inquiry example .....	2-4
Using multiple display formats .....	2-5
Multiple format possibilities .....	2-6
Non-overlapping formats: OVERLAY keyword .....	2-7
Item inquiry DDS: OVERLAY .....	2-8
Item inquiry program: OVERLAY .....	2-10
Minimizing data transfer .....	2-11
Enhance item inquiry (1 of 2) .....	2-12
Enhance item inquiry (2 of 2) .....	2-14
Item inquiry DDS: PUTOVR/OVRDTA/OVRATA .....	2-15
Item inquiry RPG: PUTOVR/OVRDTA/OVRATR .....	2-17
Machine exercise: Using OVERLAY and PUTOVR in display files .....	2-18
2.1. Working with orders.....	2-19
Work with orders .....	2-20
Window keyword .....	2-22
DDS coding for WINDOW keyword .....	2-25
WDWBORDER keyword .....	2-27
WDWTITLE keyword .....	2-29
WINDOW example: DDS .....	2-31
WINDOW example: RPG IV .....	2-32
WINDOW example: Result .....	2-33
RMVWDW keyword: Remove window .....	2-34
USRSTDSP keyword: User restore display .....	2-35
USRSTDSP user restore display: Example .....	2-36
Simple window: DDS .....	2-37
Simple window: RPG IV .....	2-38
Machine exercise: Using DDS windows .....	2-39

Unit summary .....	2-40
<b>Unit 3. Using arrays . . . . .</b>	<b>3-1</b>
Unit objectives .....	3-2
What are arrays and tables? .....	3-3
What is an array? .....	3-5
Processing data in an array .....	3-6
Defining an array .....	3-7
Summary of D-Spec keywords for arrays .....	3-9
Define, load, and access array data .....	3-11
Compile time load .....	3-13
When to use arrays .....	3-14
Naming arrays and tables .....	3-15
Moving array data (1 of 2) .....	3-16
Moving array data (2 of 2) .....	3-17
Crossfooting arrays .....	3-18
Using arrays in a loop .....	3-19
%LookUpxx .....	3-20
%LookUpxx example .....	3-21
SortA .....	3-23
BIFs and arrays: %SIZE .....	3-24
BIFs and arrays: %ELEM .....	3-25
BIFs and arrays: %SUBARR .....	3-26
Machine exercise: Array processing .....	3-27
Unit summary .....	3-28
<b>Unit 4. Using data structures and data areas . . . . .</b>	<b>4-1</b>
Unit objectives .....	4-2
What is a data structure? .....	4-3
Characteristics of data structures .....	4-5
Types of data structures .....	4-7
Defining data structures .....	4-8
Program described data structure .....	4-10
Redefining fields .....	4-11
Externally described data structure .....	4-13
Uses for data structures .....	4-15
Grouping fields .....	4-16
Subdividing fields .....	4-17
Using a data structure and named indicators .....	4-18
Qualified names .....	4-20
QUALIFIED and LIKEDS: Example .....	4-21
Compound data structures .....	4-22
LIKEREC keyword .....	4-23
Arrays within data structure .....	4-24
Using arrays within data structures .....	4-25
Pre-V5R2: Multiple occurrence data structure .....	4-26
Processing occurrences of MOD/S .....	4-27
V5R2: Array data structure .....	4-29

Array as a component of a data structure . . . . .	4-30
Multidimensional array . . . . .	4-31
Pre-V5R2: Multidimensional array . . . . .	4-32
V5R2: Multidimensional array . . . . .	4-33
Reinitializing data structures . . . . .	4-34
What is a data area? . . . . .	4-36
Uses for a data area . . . . .	4-37
Comparison of data area types . . . . .	4-38
Explicitly process a data area . . . . .	4-39
Naming a data area using a variable . . . . .	4-41
Implicitly process a data area (1 of 2) . . . . .	4-42
Implicitly process a data area (2 of 2) . . . . .	4-43
Local data area . . . . .	4-44
LDA definition, IN and OUT . . . . .	4-45
Implicit definition of LDA . . . . .	4-46
Machine exercise: Data structures/data areas . . . . .	4-47
Unit summary . . . . .	4-48
<b>Unit 5. Using subfile displays . . . . .</b>	<b>5-1</b>
Unit objectives . . . . .	5-2
5.1. Creating a subfile. . . . .	5-3
What is a subfile? . . . . .	5-4
Subfile used for inquiry . . . . .	5-5
Sample inquiry subfile application . . . . .	5-7
Sample inquiry subfile application: DDS . . . . .	5-8
Sample inquiry subfile application: Program . . . . .	5-11
Writing records to a subfile . . . . .	5-13
Writing records to a subfile: RPG IV . . . . .	5-14
Performing display I/O and control functions . . . . .	5-16
Physical display I/O: EXFMT . . . . .	5-17
Control function example: Clear subfile . . . . .	5-19
Machine exercise: Inquiry subfiles . . . . .	5-20
5.2. Adding a search argument . . . . .	5-21
Search argument with subfile: Output . . . . .	5-22
Search argument with subfile: DDS . . . . .	5-24
Search argument with subfile: Flowchart (1 of 3) . . . . .	5-26
Search argument with subfile: Flowchart (2 of 3) . . . . .	5-27
Search argument with subfile: Flowchart (3 of 3) . . . . .	5-28
Search argument with subfile: RPG program . . . . .	5-29
Other subfile keywords . . . . .	5-31
Machine exercise: Inquiry subfiles with search . . . . .	5-33
5.3. Subfile maintenance . . . . .	5-35
Considerations for subfile maintenance . . . . .	5-36
Options . . . . .	5-37
Step-by-step process to adding maintenance function . . . . .	5-38
Modular version of vendor search: DDS . . . . .	5-39
Modular version of vendor search: RPG IV (1 of 3) . . . . .	5-40
Modular version of vendor search: RPG IV (2 of 3) . . . . .	5-41

Modular version of vendor search: RPG IV (3 of 3) .....	5-43
Machine exercise: Modularize vendor subfile search .....	5-44
<b>5.4. Additional maintenance considerations .....</b>	<b>5-45</b>
Single page subfile load .....	5-46
Page+1 and Pagedown: DDS (1 of 2) .....	5-48
Page+1 and Pagedown: DDS (2 of 2) .....	5-49
Page+1 and Pagedown: RPG IV (1 of 3) .....	5-51
Page+1 and Pagedown: RPG IV (2 of 3) .....	5-52
Page+1 and Pagedown: RPG IV (3 of 3) .....	5-53
Machine exercise: Page+1 and Pagedown .....	5-54
Pageup: DDS .....	5-55
Pageup: RPG IV (1 of 4) .....	5-56
Pageup: RPG IV (2 of 4) .....	5-57
Pageup: RPG IV (3 of 4) .....	5-59
Pageup: RPG IV (4 of 4) .....	5-60
Machine exercise: Add PageUp .....	5-62
<b>5.5. Implement SFLPAG = SLFSIZ .....</b>	<b>5-63</b>
Add SFLPAG = SLFSIZ: DDS .....	5-64
Add SFLPAG = SLFSIZ: RPG IV (1 of 5) .....	5-65
Add SFLPAG = SLFSIZ: RPG IV (2 of 5) .....	5-66
Add SFLPAG = SLFSIZ: RPG IV (3 of 5) .....	5-67
Add SFLPAG = SLFSIZ: RPG IV (4 of 5) .....	5-68
Add SFLPAG = SLFSIZ: RPG IV (5 of 5) .....	5-69
Machine exercise: Add SFLPAG = SLFSIZ .....	5-70
<b>5.6. Adding file maintenance .....</b>	<b>5-71</b>
Subfile used for DB maintenance .....	5-72
Activation .....	5-73
Maintenance: Design considerations .....	5-75
Add maintenance: DDS (1 of 2) .....	5-76
Add maintenance: DDS (2 of 2) .....	5-77
Add maintenance: RPG IV (1 of 7) .....	5-78
Add maintenance: RPG IV (2 of 7) .....	5-79
Add maintenance: RPG IV (3 of 7) .....	5-80
Add maintenance: RPG IV (4 of 7) .....	5-81
Add maintenance: RPG IV (5 of 7) .....	5-82
Add maintenance: RPG IV (6 of 7) .....	5-83
Add maintenance: RPG IV (7 of 7) .....	5-85
Machine exercise: Add maintenance .....	5-86
Maintenance: Summary .....	5-87
Unit summary .....	5-88
<b>Unit 6. Managing exceptions and handling errors .....</b>	<b>6-1</b>
Unit objectives .....	6-2
<b>6.1. Using BIFs to avoid errors .....</b>	<b>6-3</b>
BIFs and errors .....	6-4
Summary of I/O BIFs .....	6-5
%Eof .....	6-6
%Equal .....	6-7

%Found .....	6-8
%Error and %Found with SETLL .....	6-9
%Open .....	6-10
Program problems .....	6-11
Exceptions and methods .....	6-12
Methods of managing exceptions .....	6-14
IBM i logic flow for error handling .....	6-16
Default Error Handler .....	6-18
Using E-Extender .....	6-19
Using the (E) extender .....	6-20
Using BIFs with opcodes .....	6-22
Using %Error to handle an error .....	6-24
INFDS: File information data structure .....	6-25
Subset of file status codes .....	6-27
E-extender, %Error, and %Status .....	6-28
%Status with WorkStn file .....	6-30
Handling a duplicate key on write to file .....	6-32
Using SETLL to avoid duplicate key error .....	6-34
Program status data structure .....	6-35
Examples of program status codes .....	6-37
What else is in the program status data structure .....	6-38
Error handling recommendations .....	6-39
Machine exercise: Error-handling BIFs .....	6-40
<b>6.2. Monitor groups.....</b>	<b>6-41</b>
What can be coded to handle this error? .....	6-42
Monitor group (1 of 2) .....	6-43
Monitor group (2 of 2) .....	6-44
Monitor group isolates error (1 of 2) .....	6-46
Monitor group isolates error (2 of 2) .....	6-47
Error handling hierarchy .....	6-48
Machine exercise: Using monitor groups .....	6-49
Unit summary .....	6-50
<b>Unit 7. Processing date and time data.....</b>	<b>7-1</b>
Unit objectives .....	7-2
What can be done? .....	7-3
Defining date, time, and timestamp fields .....	7-4
Date formats .....	7-6
Date values .....	7-8
Time formats .....	7-9
Time values .....	7-10
Specifying date and time formats .....	7-11
Define date, time, and timestamp fields .....	7-13
Externally defined date and time fields .....	7-14
Example: Externally defined dates (1 of 2) .....	7-16
Example: Externally defined dates (2 of 2) .....	7-17
Moving data to and from date, time, and timestamp .....	7-18
Testing for valid dates .....	7-20

Setting date fields to job and system dates .....	7-22
Moving to and from date and time fields (1 of 2) .....	7-23
Moving to and from date and time fields (2 of 2) .....	7-24
Date and time durations .....	7-26
Adding durations .....	7-27
Subtracting durations .....	7-29
Duration considerations (1 of 2) .....	7-31
Duration considerations (2 of 2) .....	7-32
Durations based on today's date .....	7-34
Extracting from dates, times, and timestamps .....	7-35
Comparing dates and times .....	7-36
Testing for valid values for date, time, and timestamp .....	7-37
Calculate last day of any month .....	7-38
Calculate day of week .....	7-39
Calculate age .....	7-40
Machine exercise: Using dates .....	7-41
Unit summary .....	7-42
<b>Unit 8. Using prototyped program calls.....</b>	<b>8-1</b>
Unit objectives .....	8-2
Calls in RPG IV .....	8-3
Calling another program .....	8-4
The calling program .....	8-5
The called program .....	8-7
Calling program calls called program .....	8-9
Return to calling program .....	8-10
Advantages of prototyping .....	8-11
Read-only parameters .....	8-12
Read-only parameters: Caveat .....	8-14
PR and PI with matching CONST keyword .....	8-15
Options keyword for parameter definition .....	8-17
Example prototype for QCMDEXC .....	8-19
What else can prototypes do? .....	8-20
Machine exercise: Prototyping .....	8-21
Unit summary .....	8-22
<b>Unit 9. Using subprocedures.....</b>	<b>9-1</b>
Unit objectives .....	9-2
What is a subprocedure? .....	9-3
Example: DayOfWeek subroutine .....	9-5
Example: Basic subprocedure .....	9-6
Different types of RPG IV source members .....	9-8
Major features of subprocedures .....	9-10
What are local variables? .....	9-11
Basic subprocedure .....	9-12
Invoking the subprocedure .....	9-14
P-specs and the procedure interface .....	9-15
Local variables in DayOfWeek subprocedure .....	9-17

Returning the result . . . . .	9-18
Defining the prototype . . . . .	9-19
RPG IV specification sequence . . . . .	9-21
Subprocedures calling other subprocedures . . . . .	9-23
An alternate approach . . . . .	9-24
Machine exercise: Subprocedures . . . . .	9-25
Unit summary . . . . .	9-26
<b>Unit 10. An introduction to the Integrated Language Environment. . . . .</b>	<b>10-1</b>
Unit objectives . . . . .	10-2
10.1. Terms and packaging . . . . .	10-3
What is ILE? . . . . .	10-4
Why ILE? . . . . .	10-5
10.2. Creating ILE objects . . . . .	10-9
Creating non-ILE RPG IV programs (1 of 4) . . . . .	10-10
Creating non-ILE RPG IV programs (2 of 4) . . . . .	10-11
Definitions . . . . .	10-13
Procedure (1 of 2) . . . . .	10-14
Procedure (2 of 2) . . . . .	10-15
Module (1 of 2) . . . . .	10-16
Module (2 of 2) . . . . .	10-17
Program (1 of 2) . . . . .	10-18
Program (2 of 2) . . . . .	10-19
Service program (1 of 2) . . . . .	10-21
Service program (2 of 2) . . . . .	10-22
Creating ILE RPG IV programs (3 of 4) . . . . .	10-23
Creating ILE RPG IV programs (4 of 4) . . . . .	10-24
Creating ILE programs: Bind by copy . . . . .	10-25
PEP and UEP . . . . .	10-27
CRTPGM command and ENTMOD . . . . .	10-29
Machine exercises: Creating ILE objects and bind by copy . . . . .	10-30
10.3. Service programs . . . . .	10-31
Why service programs? . . . . .	10-32
Creating a service program . . . . .	10-33
CRTSRVPGM command . . . . .	10-34
Using service programs . . . . .	10-35
CRTPGM command and BNDSRVPGM . . . . .	10-36
Creating ILE objects . . . . .	10-37
Machine exercises: Bind by reference . . . . .	10-38
10.4. Call types. . . . .	10-39
Modular programming . . . . .	10-40
Dynamic CALL versus static CALL . . . . .	10-41
Example: Dynamic CALL versus static CALL . . . . .	10-42
Static binding . . . . .	10-43
Bind by copy and bind by reference . . . . .	10-45
Why two types of static binding? . . . . .	10-47
Updating a *PGM . . . . .	10-49
Updating a *SRVPGM . . . . .	10-50

Sharing data in ILE programs . . . . .	10-51
IMPORT and EXPORT keywords . . . . .	10-53
IMPORT / EXPORT example . . . . .	10-54
ILE is much, much more . . . . .	10-55
Unit summary . . . . .	10-57
<b>Unit 11. What next? . . . . .</b>	<b>11-1</b>
Unit objectives . . . . .	11-2
What have you learned? . . . . .	11-3
To do list t . . . . .	11-4
Topics for the advanced class . . . . .	11-5
Useful references and Web sites . . . . .	11-6
Unit summary . . . . .	11-7

# Trademarks

The reader should recognize that the following terms, which appear in the content of this training document, are official trademarks of IBM or other companies:

IBM® is a registered trademark of International Business Machines Corporation.

The following are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

AS/400®	Balance®	DB2®
i5/OS®	Integrated Language Environment®	iSeries®
Language Environment®	Notes®	Operating System/400®
OS/400®	Rational®	Redbooks®
RPG/400®	System i®	WebSphere®
400®		

Adobe is either a registered trademark or a trademark of Adobe Systems Incorporated in the United States, and/or other countries.

Pentium is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Windows is a trademark of Microsoft Corporation in the United States, other countries, or both.

UNIX® is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.



# Course Description

## RPG IV Programming Intermediate Workshop for i

**Duration:** 4 days

### Purpose

This course teaches programmers who can write basic RPG IV programs using the i5(iSeries) RPG IV compiler (Version 5), additional skills and techniques.

This class is a comprehensive exposure to intermediate features and functions of RPG IV for Version 5.

### Audience

This course is the second in a series of three classes designed for programmers who are new to RPG IV. Basic programming experience using RPG IV is mandatory. The student should have attended AS06/S6197.

This course is not designed for RPG III programmers who want RPG IV. RPG III programmers should attend the Moving from RPG/400 to RPG IV class (OE85/S6126) rather than this class. RPG III programmers should review the agenda carefully before they make a decision to attend this class.

In those geographies where OE85/S6126) is not offered, RPG II programmers can attend this course, assuming they meet all prerequisites.

**Note:** This course focuses entirely on the latest features of the RPG IV for Version 5 compiler and the features that this compiler provides.

Previous techniques and the maintenance of programs written using legacy techniques are not covered in the classroom. Some additional material and the bibliography assist the students in the maintenance of legacy applications.

### Prerequisites

Before attending this course, the student should be able to:

- Use a Windows-based PC
- Run PC applications using menus, icons, tool bars, etcetera

The following skills are taught in OL49/S6149:

- Use basic i5/OS (OS/400) navigation tools, including:
  - Use and prompt CL commands
  - Use online Help
  - Manage output using WRKSPLF and related commands
  - Perform basic problem determination using DSPMSG, DSPJOB, and so forth
- Use and display i5(iSeries) print queues
- Use basic CL commands, such as WRKJOB and DSPMSG
- Use the Program Development Manager, Source Entry Utility, or RSE LPEX editor to create and maintain RPG IV source members
- Create and maintain physical and logical files

The following skills are taught in AS06/S6197:

- Write simple RPG IV version 5 programs to produce reports
- Write simple RPG IV version 5 inquiry programs that interact with displays
- Use the editor to enter and modify source programs compile RPG IV programs
- Review compilation listing, find and correct compilation errors
- Maintain existing applications written in the RPG IV (version 5) language
- Use the Debugger tool to determine the cause of incorrect results
- Use many popular RPG IV Built in Functions

Students must have attended these courses prior to attending this class, or have equivalent experience:

- ***Introduction to i5(iSeries) for New Users*** (OE98/S6108)
- ***i5(iSeries) Application Programming Facilities Workshop*** (OL49/S6149)
- ***i5(iSeries) RPG IV Version 5 Programming Fundamentals Workshop*** (AS06/S6197)

Attendance at ***i5(iSeries) Application Development using WDSC V5 for i5(iSeries)*** (AS86/S6286/S6586) is strongly recommended.

## Objectives

After completing this course, you should be able to:

- Use OVERLAY and related DDS keywords to develop efficient interactive programs
- Write interactive programs that support inquiry of subfiles
- Write interactive programs that support maintenance of subfile records
- Use arrays and data structures in RPG IV programs
- Develop RPG IV programs that anticipate and manage common errors and exceptions
- Define date and time data
- Calculate durations between two dates
- Extract month, day, or year components of date data
- Use prototyping to call other programs
- Write RPG IV subprocedures
- Write ILE modules and include those modules in program objects using bind by copy and bind by reference

## Contents

1. Welcome and administration
2. Managing display record formats
3. Using arrays
4. Using data structures and data areas
5. Using subfile displays
6. Managing exceptions and handling errors
7. Processing date and time data
8. Prototyped calls
9. Using subprocedures
10. Introduction to ILE
11. What's next?

## Curriculum relationship

This class is the second of three classes designed to develop new RPG IV programmers. It is part of the i5(iSeries) programming curriculum to support the RPG IV Version 5 compiler. Below are others that you should attend to enhance your skills using the RPG IV language after you have completed this course. Please contact your local IBM Education Center regarding the availability of these classes in your area:

- AS10/S6199 - *i5(iSeries) RPG IV Version 5 Programming Advanced Workshop*
- AS86/S6586 - *i5(iSeries) Application Development using WDSC for i5(iSeries)*
- OL37/S6137 - *Accessing the DB2 UDB for i5(iSeries) Database Using SQL*
- OL38/S6138 - *Developing i5(iSeries) Applications Using SQL*

# Agenda

## Day 1

- Unit 1: Welcome and administration
- Unit 2: Managing display record formats
- Unit 3: Using arrays
- Unit 4: Using data structures and data areas

## Day 2

- Unit 5: Using subfile displays

## Day 3

- Unit 6: Managing exceptions and handling errors
- Unit 7: Processing date and time data
- Unit 8: Using prototyped program calls

## Day 4

- Unit 9: Using subprocedures
- Unit 10: An introduction to Integrated Language Environment
- Unit 11: What's next?



# Unit 1. Welcome and administration

## What this unit is about

This unit opens the course. The instructor covers the administrative items required for this class. Students introduce themselves and discuss their individual backgrounds and expectations from the course. The instructor explains the objectives of the course as well as the agenda.

## What you should be able to do

After completing this unit, you should be able to:

- List the course objectives
- Explain what is covered during the course

# Administration



IBM i

- Student information:
  - Badges and security
  - Phones and messages
  - Facilities
  - Class hours
  - Local restaurants
  - Maps: Area and building
- Questions
- Introductions

© Copyright IBM Corporation 2009

---

Figure 1-1. Administration

AS075.0

## Notes:

# Course objectives

IBM i

After completing this course, you should be able to:

- Use OVERLAY and related DDS keywords to develop efficient interactive programs
- Write interactive programs that support inquiry of subfiles
- Write interactive programs that support maintenance of subfile records
- Use arrays and data structures in RPG IV programs
- Develop RPG IV programs that anticipate and manage common errors and exceptions
- Define date and time data
- Calculate durations between two dates
- Extract month, day, or year components of date data
- Use prototyping to call other programs
- Write RPG IV subprocedures
- Write ILE modules and include those modules in program objects using bind by copy and bind by reference

© Copyright IBM Corporation 2009

Figure 1-2. Course objectives

AS075.0

## Notes:

We continue at the point we left off at end of the AS06/AS060 class.

# Prerequisites

- Before attending this course, the student should be able to:
- Use a Microsoft Windows-based PC and run PC applications
  - Use IBM i CL commands and work with spooled output
  - Use PDM / SEU or the RSE LPEX Editor to create and maintain RPG IV source members
  - Create and maintain physical and logical files
  - Write RPG IV version 6 programs to produce reports
  - Write simple RPG IV version 6 inquiry programs that interact with displays
  - Use the editor to enter and modify source programs compile RPG IV programs
  - Review compilation listing, and find and correct compilation errors
  - Maintain existing applications written in the RPG IV (version 6) language
  - Use the Debugger tool to determine the cause of incorrect results
  - Use many popular RPG IV built-in functions

© Copyright IBM Corporation 2009

Figure 1-3. Prerequisites

AS075.0

## Notes:

You should have attended AS06/AS060 before this class.

# Agenda (1 of 2)

IBM i

## Day 1

- Unit 1: Welcome and administration
- Unit 2: Managing display record formats
  - Lab 1: Using OVERLAY and PUTOVR in display files
  - Lab 2: Using DDS Windows
- Unit 3: Using arrays
  - Lab 3: Array processing
- Unit 4: Using data structures and data areas
  - Lab 4: Data structures / Data areas

## Day 2

- Unit 5: Using subfile displays:
  - Lab 5: Inquiry subfiles
  - Lab 6: Inquiry subfiles with search
  - Lab 7: Modularize vendor subfile search
  - Lab 8: Page +1 and Pagedown
  - Lab 9: Add PageUp
  - Lab 10: Add SFLPAG = SFLSIZ
  - Lab 11: Add maintenance

© Copyright IBM Corporation 2009

Figure 1-4. Agenda (1 of 2)

AS075.0

## Notes:

## Agenda (2 of 2)

IBM i

### Day 3

- Unit 6: Managing exceptions and handling errors
  - [Lab 12: Error handling and BIFs](#)
  - [Lab 13 : Using monitor groups](#)
- Unit 7: Processing data and time data
  - [Lab 14: Using dates](#)
- Unit 8: Using prototyped program calls
  - [Lab 15: Prototyping](#)

### Day 4

- Unit 9: Using subprocedures
  - [Lab 16: Subprocedures](#)
- Unit 10: An introduction to Integrated Language Environment
  - [Lab 17: Creating ILE objects](#)
  - [Lab 18: Bind by copy](#)
  - [Lab 19: Bind by reference](#)
- Unit 11: What's next?

© Copyright IBM Corporation 2009

---

Figure 1-5. Agenda (2 of 2)

AS075.0

### Notes:

# Introductions

IBM i

- Your name and city
- Company or organization name
- Previous programming experience
- Expectations



© Copyright IBM Corporation 2009

Figure 1-6. Introductions

AS075.0

## Notes:

Introduce yourself. We are very interested in your programming background and the RPG IV programming that you did between the first class (AS06/AS060) and this one.



# Unit 2. Managing display record formats

## What this unit is about

This unit describes the use of special DDS keywords that can make handling display formats easier as well as more efficient.

## What you should be able to do

After completing this unit, you should be able to:

- Create display file DDS that supports OVERLAY, PUTOVR, OVRATR, and OVRDTA keywords
- Explain the purpose of OVERLAY, PUTOVR, OVRATR, and OVRDTA keywords
- Code RPG IV interactive programs that take advantage of display files that contain display files with the OVERLAY, PUTOVR, OVRATR, and OVRDTA keywords
- Design windows using DDS and write programs to use windows

## How you will check your progress

- Machine exercises
- Given an existing inquiry program that uses several display formats, the students add OVERLAY and OVRDTA keyword support to enhance the presentation of the data.
- The students modify the above program to display messages in a window using DDS window keywords.

# Unit objectives

IBM i

After completing this unit, you should be able to:

- Create display file DDS that supports OVERLAY, PUTOVR, OVRATR, and OVRDTA keywords
- Explain the purpose of OVERLAY, PUTOVR, OVRATR, and OVRDTA keywords
- Code RPG IV interactive programs that take advantage of display files that contain display formats with the OVERLAY, PUTOVR, OVRATR, and OVRDTA keywords
- Design windows using DDS and write programs to use windows

© Copyright IBM Corporation 2009

---

Figure 2-1. Unit objectives

AS075.0

## Notes:

# Inquiry example: Display file DDS

```

ITEMINQ.DSPF X
Line 33 Column 1 Insert
....AAN01N02N03..Name+++++RLen++TDpBLinPosFunctions+++++++
000100 A REF(*LIBL/ITEM_PF) CA03(03) INDARA
000400 A R PROMPT
000500 A PGMNAM 10A 0 3 7
000600 A 3 35'Item Inquiry' COLOR(WHT)
000800 A 3 64DATE EDTCDE(Y)
001000 A 8 20'Item Number . . . . :'
001100 A ITMNBR R D I 8 45
001200 A 40 ERMSG('Item not found on file - pl-
001300 A ease correct' 40)
001400 A 20 7'Press Enter to continue'
001500 A 21 7'F3=Exit' COLOR(BLU)
001800 A R DETAIL
002000 A PGMNAM 10A 0 3 7
002100 A 3 35'Item Inquiry' COLOR(WHT)
002300 A 3 64DATE EDTCDE(Y)
002500 A 8 20'Item Number . . . . :'
002600 A ITMNBR R 0 8 45
002700 A 9 20'Description . . . . :'
002800 A ITMDESCR R 0 9 45
002900 A 11 20'Quantity on hand . . . . :'
003000 A ITMQTYOH R 0 11 45EDTCDE(Z)
003100 A 12 20'DSPATR(HI)
003200 A 13 20'Quantity on order . . . . :'
003300 A ITMQTYOO R 0 12 45EDTCDE(Z)
003400 A 14 20'DSPATR(HI)
003500 A 14 20'Supplier Number . . . . :'
003600 A VNDNBR R 0 14 45
003700 A 15 20'Supplier Catalogue No . :'
003800 A ITMVHDCAT#R 0 15 45
003900 A 17 7'** NOTE ** The total quantity avai-
004000 A lable is low (<20) - reorder'
004100 A 21 7'F3=Exit F12=Cancel' COLOR(BLU)
004200 A

```

© Copyright IBM Corporation 2009

Figure 2-2. Inquiry example: Display file DDS

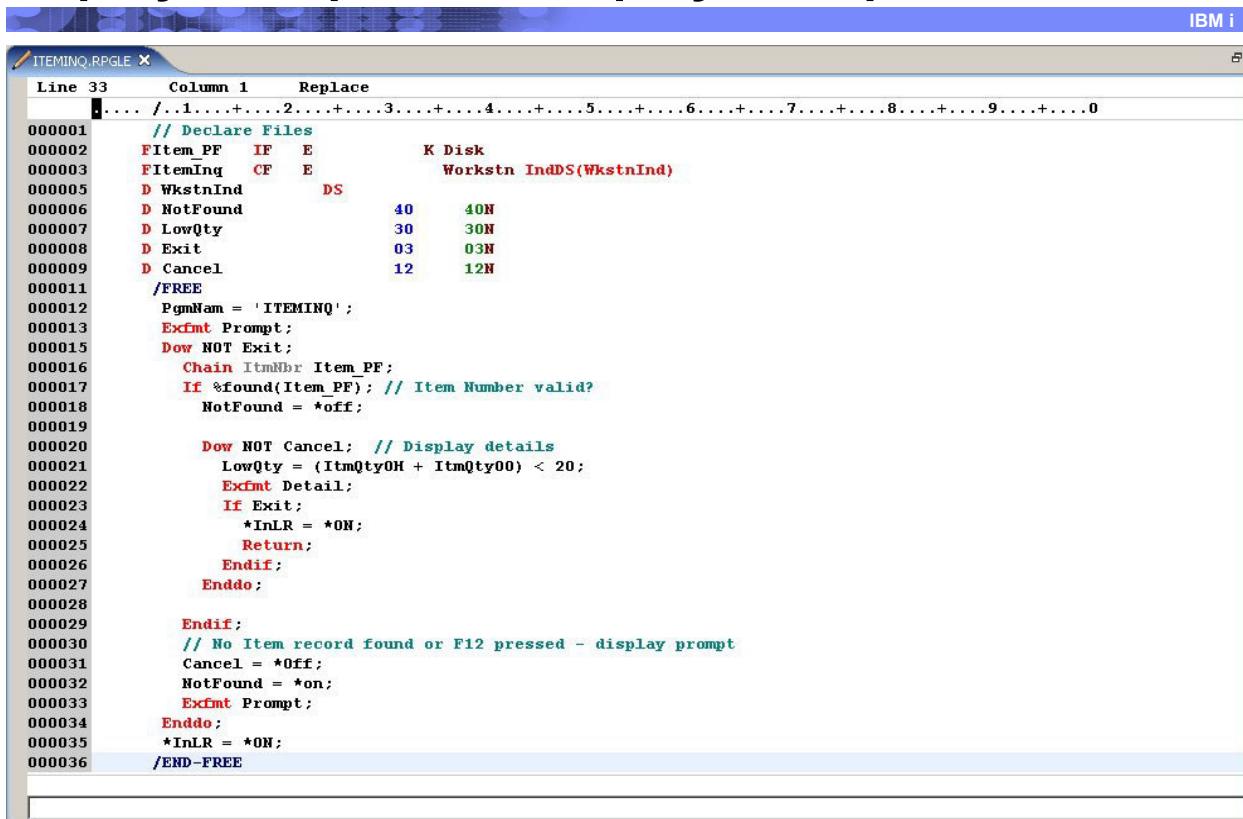
AS075.0

## Notes:

This is a typical inquiry display file. There are two formats:

- Prompt:** Expects an item number and displays a message for an invalid item number.
- Detail:** For valid items, displays information from the item record retrieved.

# Inquiry example: Item inquiry example



```

ITEMINQ.RPGLE X
Line 33    Column 1      Replace
000001 // Declare Files
000002 FItem_PF  IF   E          K Disk
000003 FItemInq  CF   E          Workstn IndDS(WkstnInd)
000005 D WkstnInd       DS
000006 D NotFound        40     40N
000007 D LowQty         30     30N
000008 D Exit            03     03N
000009 D Cancel          12     12N
000011 /FREE
000012 PgmNam = 'ITEMINQ';
000013 ExFmt Prompt;
000015 Dow NOT Exit;
000016   Chain ItmNbr Item_PF;
000017   If %found(Item_PF); // Item Number valid?
000018     NotFound = *off;
000019
000020   Dow NOT Cancel; // Display details
000021     LowQty = (ItmQty0H + ItmQty00) < 20;
000022     ExFmt Detail;
000023     If Exit;
000024       *InLR = *ON;
000025       Return;
000026     Endif;
000027   Enddo;
000028
000029 Endif;
000030 // No Item record found or F12 pressed - display prompt
000031 Cancel = *Off;
000032 NotFound = *on;
000033 ExFmt Prompt;
000034 Enddo;
000035 *InLR = *ON;
000036 /END-FREE

```

© Copyright IBM Corporation 2009

Figure 2-3. Inquiry example: Item inquiry example

AS075.0

## Notes:

This is the program that supports the DDS display file. Two formats are controlled by the use of function keys. The program has logic to support cancel (F12) as well as exit (F3).

# Using multiple display formats

IBM i

- Can separate full format into parts
- Reduces amount of data sent

```

----- Header -----
ITEMINQ           Item Inquiry      6/22/00
                  Prompt

----- Detail -----
Item Number . . . . . : 10025
Description . . . . . Heavy duty stapler

Quantity on hand . . . . .   5
Quantity on order . . . . .

Supplier Number . . . . . : 10050
Supplier Catalogue No . . : 99932

** NOTE ** The total quantity available is low (<20) - reorder
----- Footer -----
Press Enter to continue
F3=Exit

----- Item not found on file - please correct -----

```

© Copyright IBM Corporation 2009

Figure 2-4. Using multiple display formats

AS075.0

## Notes:

Working with multiple display formats for inquiry or update applications is not much different from working with single display formats. The main differences are that you must control which format is being displayed and read as well as ensure data field integrity. We look at more DDS keywords that have a dramatic impact on your displays and the programs that use them.

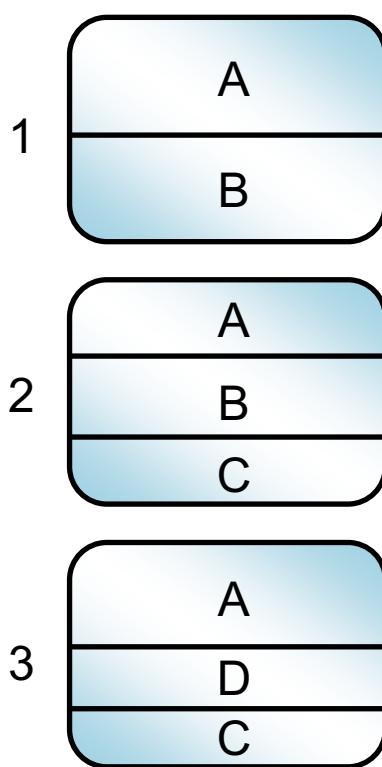
At first glance, creating a full single format for each transaction might make more sense. However, building the final display from multiple formats can be much more effective in terms of performance as well as provide you with a lot of flexibility and reuse in your code.

**Tip:** By default, the workstation display is cleared by each new format unless you override it.

# Multiple format possibilities

IBM i

## Non-Overlapping Formats



© Copyright IBM Corporation 2009

Figure 2-5. Multiple format possibilities

AS075.0

## Notes:

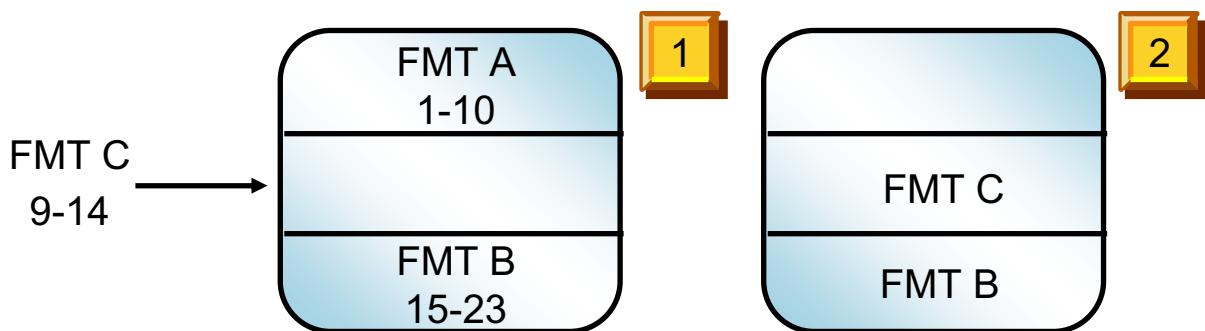
Inquiry, data entry, subfiles, online help and update are valid applications that could benefit from multiple display format use.

When a new display format is written to the device, any previously written display format is erased (by default). If two or more formats are to share the display, we need new DDS keywords to make it work.

## Non-overlapping formats: OVERLAY keyword

IBM i

- Record-level keyword.
- Reduces need to resend formats.
- Does not clear the display device.
- Formats touched by overlaying format are erased automatically.



© Copyright IBM Corporation 2009

Figure 2-6. Non-overlapping formats: OVERLAY keyword

AS075.0

### Notes:

OVERLAY is a record-level keyword used to specify that the record format you are defining should appear on the display without the entire display being erased first.

When using OVERLAY, it is important that none of the formats overlap line numbers. If they do, the format being overlaid is erased entirely.

# Item inquiry DDS: OVERLAY

IBM i

```

000100 - A
000200>> 7 A
000201 - A
000300 ***
000400>> 1 A      R HEADER      PGMNAME      10A  0  3  7
000500>> 6 A      3 35'Item Inquiry' COLOR(WHT)
000600 - A
000700 - A
000800 ***
000900>> 2 A      R PROMPT
001000>> 5 A      ITMNBR      R      D  I  8 20'Item Number . . . . :'
001100>> 6 A      8 45
001200 - A 48
001300 - A
001400 ***
001500>> 4 A      R DETAIL
001600>> 5 A      ITMDESCR R      9 20'Description . . . . :'
001700>> 6 A      0 9 45
001800 - A
001900 - A
002000 - A 38
002100 - A
002200 - A
002300 - A 38
002400 - A
002500 - A
002600 - A
002700 - A
002800 - A 38
002900 - A
003000 ***
003100>> 3 A      R FOOTER
003200>> 5 A      20 7'Press Enter to continue'
003300>> 6 A      21 7'F3=Exit'
003400 - A      COLOR(BLU)

REF(*LIBL/ITEM_PF)
CA93(83 'Exit')
INDARA

OVERLAY
3 35'Item Inquiry' COLOR(WHT)
3 64DATE EDTCDE(Y)

OVERLAY
8 20'Item Number . . . . :'
8 45
ERRMSG('Item not found on file - please correct' 48)

OVERLAY
9 20'Description . . . . :'
9 45
11 20'Quantity on hand . . . . :'
11 45EDTCDE(Z)
DSPATR(HI)
12 20'Quantity on order . . . . :'
12 45EDTCDE(Z)
DSPATR(HI)
14 20'Supplier Number . . . . :'
14 45
15 20'Supplier Catalogue No . . . :'
15 45
17 7'** NOTE ** The total quantity available is low (<20) - reorder'

OVERLAY
20 7'Press Enter to continue'
21 7'F3=Exit'
COLOR(BLU)

```

© Copyright IBM Corporation 2009

Figure 2-7. Item inquiry DDS: OVERLAY

AS075.0

## Notes:

We modify our display file for the ITEMINQ example slightly to allow formats to be written to the screen in sequence without the display being cleared first. We have modified our DSPF to include the four formats that we noted in the previous visual:

- HEADER format:** Used for prompting in stand-alone mode as well as in conjunction with the details display. Also handles display of the error message at the bottom of the display.
- PROMPT format:** To input the item number and display an error message.
- FOOTER format:** To display keyboard instructions.
- DETAIL format:** To display information retrieved from the record when it is found and the low stock message.

Notice that:

- The OVERLAY keyword is specified in the DETAIL, PROMPT, and FOOTER formats record at the record level as they overlay the HEADER format.

2. The line numbers are checked carefully so that we can overlay the display without *overwriting* another record. If necessary, we could have changed lines that would have overlapped.

Normally, the entire display is deleted on each output operation. All records on the display with fields that partially or completely overlap fields in this record are deleted before the OVERLAY record is displayed; all others remain on the display and are not changed in any way. A record already on the display is deleted even if fields specified in the record format are not selected for display.

3. We dropped F12 because we no longer have to return to the prompt format from the detail format; F3 is still enabled for both formats.

# Item inquiry program: OVERLAY

IBM i

```

000001 // Declare Files
000002 FItem_PF IF E K Disk
000003 FItemInqOU CF E Workstn IndDS(WkstnInd)
000004
000005 // Map indicators in DSPF to named indicators
000006 D WkstnInd DS
000007 D NotFound 40 40N
000008 D LowQty 30 30N
000009 D Exit 03 03N
000010
000011 /FREE
000012 PgmNam = 'ITEMINQOU';
000013 Write Header; // Write to buffer
000014 Write Footer; // Write to buffer
000015>>1 ExFmt Prompt; // Write to buffer; write buffer to display; enable read
000016
000017>>2 Dow NOT Exit;
000018   Chain ItmNbr Item_PF;
000019   NotFound = Not %Found(Item_PF); // Set indicator for record not found
000020
000021   If %Found(Item_PF); // Item Number valid?
000022     LowQty = (ItmQty0H + ItmQty00) < 20; // Set Indicator for Qty < Minimum
000023     Write Detail; // Write to buffer
000024   EndIf;
000025
000026   // Display prompt with error or not
000027   ExFmt Prompt; // Write to buffer; write buffer to display; enable read
000028 Enddo;
000029 // F3 pressed; user wants to exit program
000030 *InLR = *on;
000031 /END-FREE

```

© Copyright IBM Corporation 2009

Figure 2-8. Item inquiry program: OVERLAY

AS075.0

## Notes:

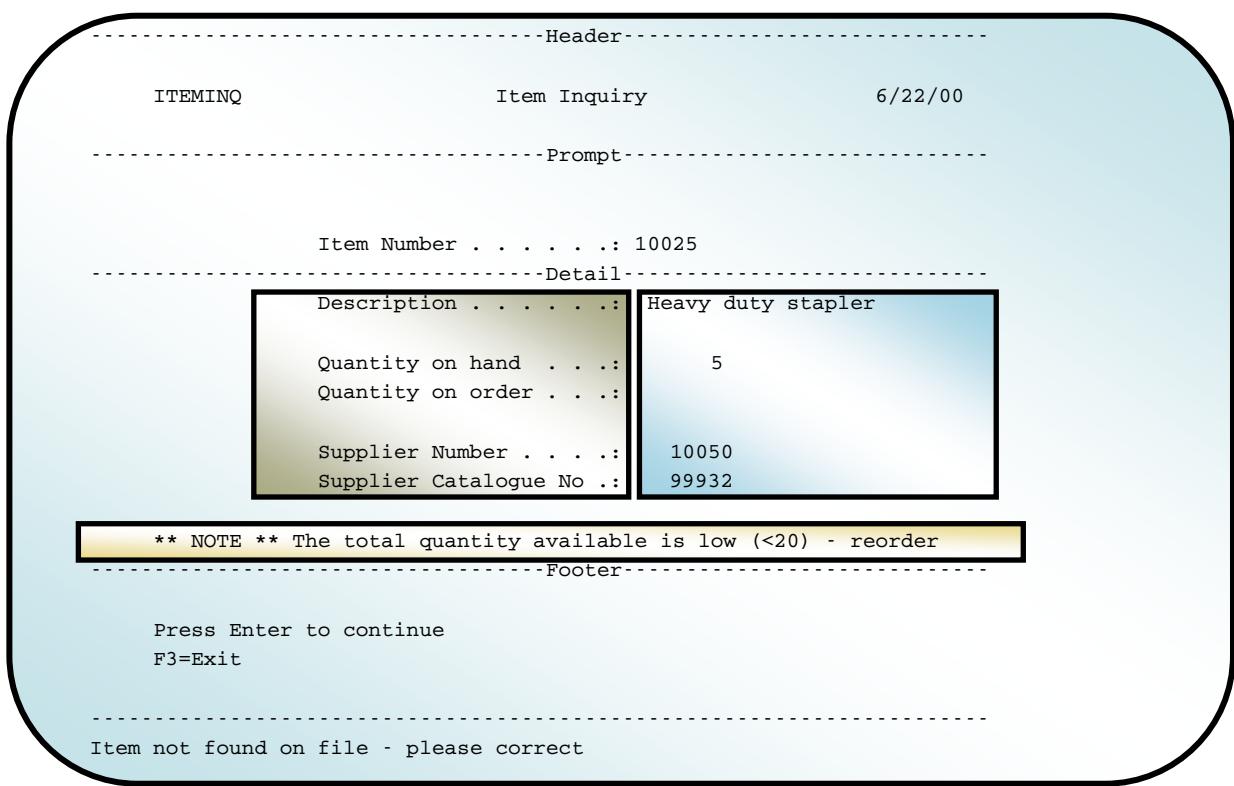
In the RPG program, additional code is required to support the HEADER and FOOTER records and to perform the overlay. The way that this is done is that we write the HEADER and FOOTER formats before we EXFMT the PROMPT format.

Now we are also displaying the prompt format with the DETAIL format:

1. We prompt for the next item on the DETAIL format. Because our program must READ the PROMPT format, we WRITE the DETAIL format and the EXFMT the PROMPT format. Thus, our program waits for us to enter the next item number and to press the Enter key.
2. Because we have *combined* the display formats of the application, we have enabled only F3. Thus, we dropped the logic that handled F12. Instead, we always EXFMT the PROMPT format immediately after WRITING the DETAIL format. The error message still displays at the bottom of the display even after we have displayed the DETAIL format for the first time.

# Minimizing data transfer

IBM i



© Copyright IBM Corporation 2009

Figure 2-9. Minimizing data transfer

AS075.0

## Notes:

OVERLAY allowed us to design and to create our display formats in a modular fashion.

As you look at the display in the visual, you notice that certain information is constant in nature while other data is variable.

For example, if we just consider the DETAIL format, the shaded area on the left is constant while the fields that come from the data file are variable. Also, the low stock message is constant in value but we may or may not want it to display depending on an indicator.

But, we cannot use OVERLAY because it works with formats laid out on horizontal lines. We want to work some elements of the format that are arranged vertically.

Next, we discuss several DDS keywords that address our need to handle this situation, and therefore, minimize the amount of data transferred between the system and the display.

# Enhance item inquiry (1 of 2)

IBM i

```

0....+...10....+...20....+...30....+...40....+...50....+...60....+...70...
-
-
-
- 1 ITEMINQ2           Item Inquiry          05/10/02
-
5
-
-
- 2 Item Number . . . . . : 00000
-
10
-
-
-
-
15
-
-
-
-
-
20 3 Press Enter to continue
- F3=Exit

```

© Copyright IBM Corporation 2009

Figure 2-10. Enhance item inquiry (1 of 2)

AS075.0

## Notes:

In an earlier example, we enhanced our Item Inquiry display file and RPG IV program to support OVERLAY of record formats. Using OVERLAY reduces the amount of data that has to be sent to the workstation.

Now, let us do the same thing with data. In our DDS, we can use certain keywords to specify that only data that has been *changed* should be sent to or from the program.

To accomplish this, we discuss these new DDS keywords:

- PUTOVR
- OVRDTA

We are still working with the four formats that we used in the OVERLAY example:

1. **HEADER** format: Used for prompting in stand-alone mode as well as in conjunction with the details display. Also handles display of the error message at the bottom of the display.
2. **PROMPT** format: To input the item number and display an error message.

3. **FOOTER** format: To display keyboard instructions.
4. **DETAIL** format: To display information retrieved from the record when it is found and the low stock message. We look at the DETAIL format next.

## Enhance item inquiry (2 of 2)

IBM i

```

0....+...10....+...20....+...30....+...40....+...50....+...60....+...70.

-
-
-
- 1      ITEMINQ2          Item Inquiry           05/10/02
-
5
-
-
- 4      Item Number . . . . . : 20001
-       Description . . . . . : Telephone, one line
10
-       Quantity on hand . . . . : 10
-       Quantity on order . . . . : 6
-
-       Supplier Number . . . . . : 10010
15       Supplier Catalog No . . . : T9939XP
-
-       ** NOTE ** The total quantity available is low (<20) - reorder
-
-
20 3      Press Enter to continue
-       F3=Exit
-
```

© Copyright IBM Corporation 2009

Figure 2-11. Enhance item inquiry (2 of 2)

AS075.0

### Notes:

We are still working with the four formats that we used in the OVERLAY example:

1. **HEADER** format: Used for prompting in standalone mode as well as in conjunction with the details display. Also handles display of the error message at the bottom of the display.
2. **PROMPT** format: To input the item number and display an error message.
3. **FOOTER** format: To display keyboard instructions.
4. **DETAIL** format: To display information retrieved from the record when it is found and the low stock message.

# Item inquiry DDS: PUTOVR/OVRDTA/OVRATA

IBM i

```

000100 A REF(*LIBL/ITEM_PF)
000101 A CA03(03)
000102 A INDARA
000200 **
000300 A R HEADER PGMNAM 10A I 3 7
000400 A 0 3 35' Item Inquiry' COLOR(WHT)
000500 A 3 64DATE EDTCDE(Y)
000600 A
000700 **
000800 A R PROMPT OVERLAY PUTOVR
000900 A ITMNBR R D B 8 20'Item Number . . . . .'
001000 A 8 45OVRDTA
001100 A 40 ERRMSG('Item not found on file - please correct' 40)
001200 A
001300 **
001400 A R DETAIL OVERLAY PUTOVR
001500 A ITMDESCR R 0 9 20'Description . . . . .'
001600 A 9 45OVRDTA
001700 A 11 20'Quantity on hand . . . . .'
001800 A 0 11 45OVRDTA EDTCDE(2)
001900 A 12 20'DSPATR(HI)'
002000 A 12 45OVRDTA EDTCDE(2)
002100 A 14 20'DSPATR(HI)'
002200 A 14 45OVRDTA
002300 A 15 20'Supplier Number . . . . .'
002400 A 15 45OVRDTA
002500 A 17 20'Supplier Catalog No . . . . .'
002600 A 17 7 '** NOTE ** The total quantity available is low (<20) - reorder'
002700 A DSPATR(HI)
002800 A DSPATR(ND)
002900 A 0 38 OVRATR
003000 A N38
003100 A
003200 **
003300 A R FOOTER OVERLAY
003400 A 20 7'Press Enter to continue'
003500 A 21 7'F3=Exit' COLOR(BLU)

```

© Copyright IBM Corporation 2009

Figure 2-12. Item inquiry DDS: PUTOVR/OVRDTA/OVRATA

AS075.0

## Notes:

There are some new keywords that we are using with this example:

**PUTOVR** allows the override of display attributes or data contents. By using **PUTOVR**, you can reduce the amount of data sent to the display. **PUTOVR** is a record-level keyword that permits the program to override the data contents, attributes or both of fields that are already displayed at the workstation. This keyword allows you to reduce the amount of data sent to the display device when **changing** the data or the attribute of displayed information.

**OVRDTA** is used with the **PUTOVR** keyword to override the existing data contents of a field or record already on the display. When **OVRDTA** is specified at both the record and field level, the field-level specification is used for that field.

**OVRATR** overrides the display attributes of a field on the screen. This field- or record-level keyword is *used in conjunction with the PUTOVR keyword* to override the existing display attributes of a field or record already on the display.

The **OVRDTA** keyword can be used with the **OVRATR** keyword on the same field or record.

The **OVRDTA** and **OVRATR** field-level keywords are used with **PUTOVR** for a particular format.

In our example, we have shown **OVRDTA** and **OVRATR** specified at field-level. Both keywords could have been defined at format-level. (For both PROMPT and DETAIL formats we have specified **OVRDTA** for every field.)

An output operation with the **OVRDTA** keyword in effect includes the function of the **OVRATR** keyword to override display attributes, as well as data contents.

# Item inquiry RPG: PUTOVR/OVRDTA/OVRATR

IBM i

```

000001 // Declare Files
000002 FItem_PF IF E K Disk
000003> FItemInq2 CF E Workstn IndDS(WkstnInd)
000004
000005 // Map indicators in DSPF to named indicators
000006 D WkstnInd DS
000007 D NotFound 40 40N
000008 D LowQty 30 30N
000009 D Exit 03 03N
000010
000011 /FREE
000012 PgmNam = 'ITEMINQOV';
000013 Write Header; // Write to buffer
000014 Write Footer; // Write to buffer
000015 ExFmt Prompt; // Write to buffer; write buffer to display; enable read
000016
000017 Dow NOT Exit;
000018 Chain ItmNbr Item_PF;
000019 NotFound = Not %Found(Item_PF); // Set indicator for record not found
000020
000021 IF %Found(Item_PF); // Item Number valid?
000022 LowQty = (ItmQtyOH + ItmQtyOO) < 20; // Set Indicator for Qty < Minimum
000023 Write Detail; // Write to buffer
000024 EndIf;
000025
000026 // Display prompt with error or not
000027 ExFmt Prompt; // Write to buffer; write buffer to display; enable read
000028 Enddo;
000029 // F3 pressed; user wants to exit program
000030 *InLR = *on;
000031 /END-FREE

```

© Copyright IBM Corporation 2009

Figure 2-13. Item inquiry RPG: PUTOVR/OVRDTA/OVRATR

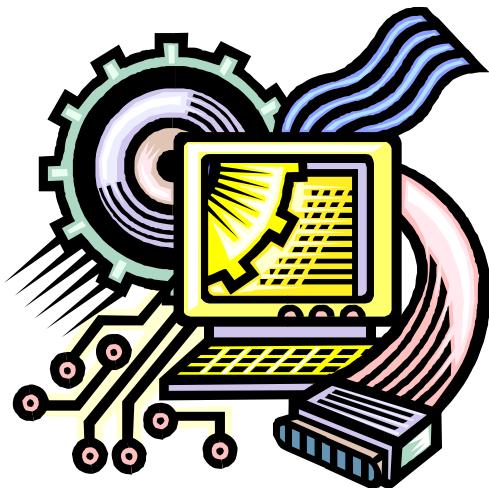
AS075.0

## Notes:

In the RPG IV program, all that changes is the name of the display file. The logic does not need to be modified. It is exactly the same as the **IteminqOV** program.

# Machine exercise: Using OVERLAY and PUTOVR in display files

IBM i



© Copyright IBM Corporation 2009

Figure 2-14. Machine exercise: Using OVERLAY and PUTOVR in display files

AS075.0

## Notes:

Perform the machine exercise “Using OVERLAY and PUTOVR in display files.”

## 2.1. Working with orders

# Work with orders

IBM i

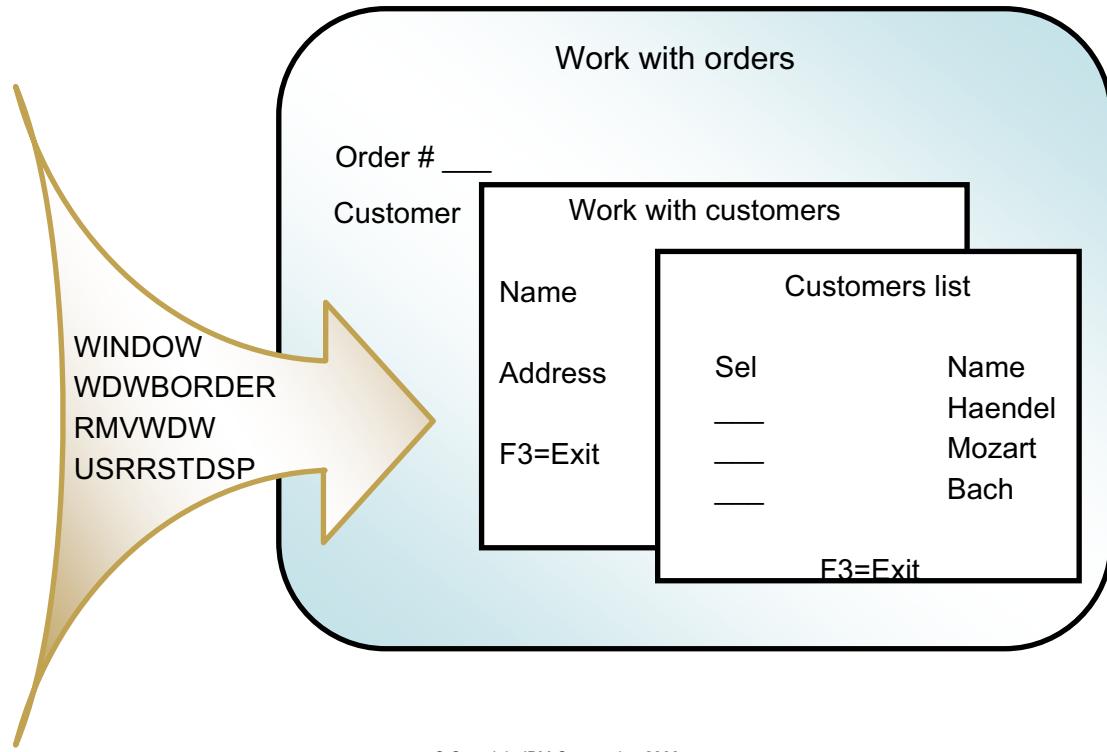


Figure 2-15. Work with orders

AS075.0

## Notes:

A window is a special record format, smaller than the actual display size, which overlays only part of the display. Three keywords allow you to define a window:

**WINDOW**

**WDWBORDER**

**WDWTITLE**

By using these keywords, you can display up to 12 windows at a time. Each window interacts with your application program as though it was a single record format with no relationship to other windows. The most recently written window is the *active window*. Your application dialogs with it and that dialog must be resolved in some fashion before interfacing with any other window.

Another keyword, **RMVWDW**, allows you to remove all existing windows on a display prior to displaying another. **USRSTDSP** means that the program is responsible for redisplaying any underlying formats when a window is displayed.

Think about the use of WINDOW over CLRL(\*NO) for overlaying formats. Several differences are that:

- Display attributes are not affected
- Underlying format does not show

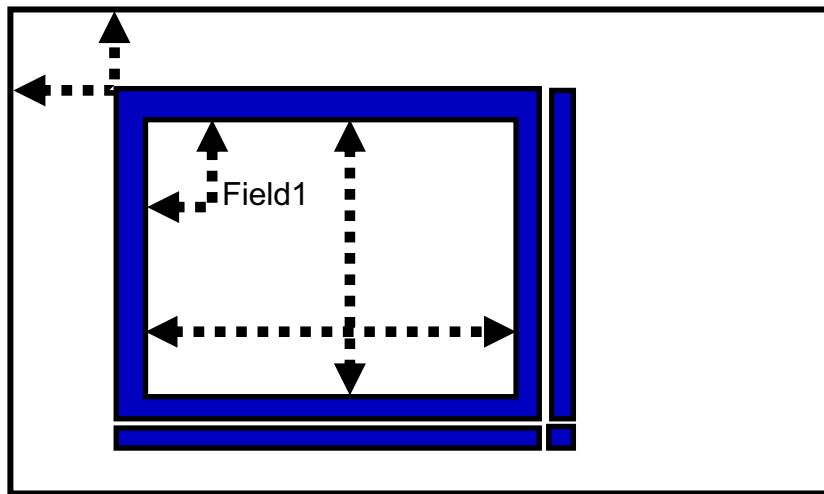
SDA can be used to assist you to create the DDS for your windows. Windows are easily coded using the SEU or LPEX editor. Read the reference manual for information on how to use SDA.

**References:**

- *ADTS for AS/400 Screen Design Aid*, Chapter 9, Creating a Window

# Window keyword

IBM i



`WINDOW(init. row    init. col.    # of rows    # of cols)`  
`WINDOW(record-format-name)`

© Copyright IBM Corporation 2009

Figure 2-16. Window keyword

AS075.0

## Notes:

**WINDOW** is a record-level keyword that names a format that can be displayed in a window. There are two ways to use this keyword:

- **WINDOW DEFINITION RECORD:** The window is defined by specifying its location and size. Although the size of the window is fixed, you may alter the location of the window on the display. You simply change the coordinates in your program.
- **WINDOW REFERENCE RECORD:** The window is defined by specifying the name of a record which contains the window's location and size.

The parameters specify:

- Initial Row of upper-left corner

This is a constant or field that contains the line number of the upper-left corner of the window border. If a variable is used, it must be defined as a signed numeric field (maximum length 3 0, type S) used as a program-to-system field (usage P).

Using this technique, your program can set a variable and pass this information to the system.

- Initial Column of upper-left corner

This is a constant or field that contains the column number that starts the upper-left corner of the window border. If a variable is used, it must be defined as a signed numeric field (maximum length 3 0, type S) used as a program-to-system field (usage P).

- Total number of lines in the window

This value cannot be greater than the number of lines on the display minus 2, because the first and last lines are used by the upper and lower borders.

The last line of the window is reserved for the message line; therefore, it cannot contain any fields.

- Total number of columns in the window

The number of columns cannot be greater than the total number of columns on the display minus 4, since the right and left borders also require one byte inside the window.

- Message Line: Possible values are \*MSGLIN/\*NOMSGLIN with a default of \*MSGLIN.

- **\*MSGLIN:** The message line is displayed in the window. If the message is longer than the available columns, it is truncated.

- **\*NOMSGLIN:** The message line is displayed as specified with the MSGLOC keyword. If nothing is specified, it is located on either line 24 or 27 of the display.

When windows exist on a display and \*MSGLIN is specified on the WINDOW keyword for the window, any error messages are displayed on the last usable line of the active window. The last usable line in the window is reserved for error messages. If the error message is longer than the line, it is truncated to fit. When windows exist on a display and \*NOMSGLIN is specified on the WINDOW keyword for the window, any error messages are displayed at the bottom of the display or the location defined by the MSGLOC keyword.

Help is available through the Help key for error messages that are displayed in windows.

The first two parameters can be replaced with the special value \*DFT. This tells the system to assign the location of the window that is based on the cursor position and the size of the window.

Lower-right window coordinates and field positions can be calculated as follows:

- Lower border line = upper border line + window lines + 1
- Right border column = left border column + window columns + 3
- Actual field line = upper border line + field line number
- Actual field column = left border column + field column + 1

The position of a field in a window is relative to the first valid position within the window. The first valid position is one line below the upper border and two columns to the right of the left border.

# DDS coding for WINDOW keyword

IBM i

---

\*\* Fixed position

```
A      R WINDOW1
A                      WINDOW(07 20 10 40)
```

\*\* Program positioning (Message line on main display)

```
A      R WINDOW2
A
A      ROW      3S 0P
A      COL      3S 0P
A                      WINDOW(&ROW &COL 10 40 *NOMSGLIN)
```

\*\* System positioning by Cursor location (Cursor not restricted)

```
A      R WINDOW3
A
A                      WINDOW(*DFT 10 40 *NORSTCSR)
```

\*\* Window referencing

```
A      R WINDOW4
A
A                      WINDOW(WINDOW1)
```

© Copyright IBM Corporation 2009

Figure 2-17. DDS coding for WINDOW keyword

AS075.0

## Notes:

As an introduction to DDS coding to support windows, this example shows some different ways of specifying the **WINDOW** keyword.

Note the use of the **\*NORSTCSR** parameter.

To position the cursor in a window, use the CSRLOC and DSPATR(PC) keyword in the same way you would for a full screen display. The cursor is positioned relative to the upper left corner of the usable area of the window.

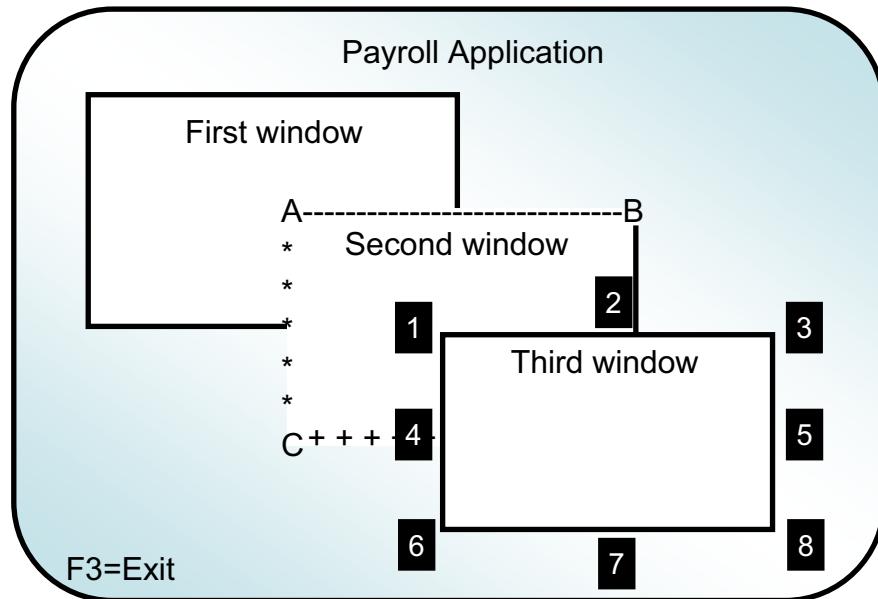
If **\*RSTCSR** is specified on the **WINDOW** keyword and the cursor is moved outside the usable area of the active window, only the Print and Home command function (CF) keys are active. If the workstation user presses any other command function (CF) key, the alarm sounds. Then the cursor is moved back to its position for the previous write operation.  
**\*RSTCSR** is the default of the **WINDOW** keyword.

On displays that support an enhanced interface for non programmable workstations, the cursor can be moved out of a window only with a mouse when **\*RSTCSR** is specified.

If **\*NORSTCSR** is specified with the **WINDOW** keyword, the user can move the cursor out of the active window and use any command function (CF) or command attention (CA) key.

# WDWBORDER keyword

IBM i



```
WDWBORDER(color display-attributes characters)
```

```
WDWBORDER((*COLOR WHT) (*DSPATR RI) (*CHAR ' '))
```

© Copyright IBM Corporation 2009

Figure 2-18. WDWBORDER keyword

AS075.0

## Notes:

This keyword is used to specify the color at the file or record level. It defines the display attributes and the characters used for the window border. The format of the keyword is:

```
WDWBORDER (color display-attributes characters)
```

The color parameter makes your border a specific color. If none is specified, the default color is blue. The syntax of the parameter is (**\*COLOR value**).

The display attributes parameter allows you to designate certain attributes for the border. The syntax is:

```
*DSPATR value1 value2 value3...
```

If more than one attribute is specified for the border, attributes are combined for the entire border.

For the characters parameter, a string of eight characters is supplied in the following order:

1. Left upper corner
2. Upper border

3. Right upper corner
4. Left border
5. Right border
6. Left lower corner
7. Lower border
8. Right lower corner

When nothing is specified, the character defaults are:

- **period (.)** For the upper left and right corners and for the upper and the lower borders.
- **colon (:)** For the lower left and right corners and for the right and left borders.

The **WDWBORDER** keyword is optional. Following is an example of the **WDWBORDER** keyword:

```
WDWBORDER ( (*COLOR BLU) (*DSPATR RI) (*CHAR '+-+| |+-+') )
```

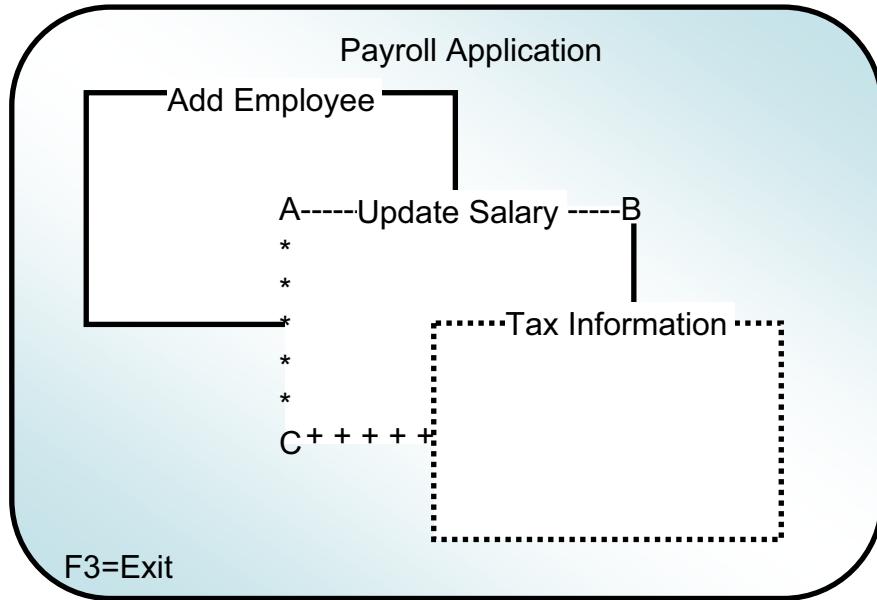
**Recommendation:**

Most PC-based 5250 emulation programs produce an acceptable default presentation of the window border. For example, the IBM Personal Communications program shows a solid blue border, with drop-shadows down the right-hand side and along the bottom of the border.

If using the WDWBORDER keyword, try to make your borders as simple but attractive as possible. Where the default is unacceptable, we recommend that (**\*COLOR WHT**) (**\*DSPATR RI**) (**\*CHAR '     '**) be used to obtain a reasonable appearance.

# WDWTITLE keyword

IBM i



```
WDWTITLE(text color attr horizontal-pos vertical-pos)
```

```
WDWTITLE((*TEXT 'Add Employee') *CENTER *TOP)
```

© Copyright IBM Corporation 2009

Figure 2-19. WDWTITLE keyword

AS075.0

## Notes:

This record-level keyword is used to specify the color and display attributes for a title. The title is placed in the upper or lower border of a window. The format of the keyword is:

```
WDWTITLE (text color attr horizontal-position vertical-position)
```

The text parameter specifies the text that is placed in the border of the window. The length should be less than or equal to the window-positions parameter of the corresponding WINDOW definition. It can be specified in one of two ways:

**((\*TEXT FirstWindow))**

**((\*TEXT &FIELD))** where &FIELD is a program-to-system field.

If a field is used, it must be specified in the window record and defined as a character field with usage **P**.

The color parameter specifies the color of the title text on a color display. It is specified as:

```
((*TEXT FirstWindow) (*COLORWHT))'
```

Valid colors are:

- BLU blue
- GRN green
- WHT white
- RED red
- TRQ turquoise
- YLW yellow
- PNK pink

The attribute parameter specifies the display attributes of the title text. If more than one display attribute is used, they are combined to form one attribute. It takes the form:

```
( (*TEXT FirstWindow') (*COLORWHT) (*DSPATTRRI) )'
```

Valid display attributes are:

- BL blink
- CS column separator
- HI high intensity
- ND non display
- RI reverse image
- UL underline

**\*CENTER/\*LEFT/\*RIGHT:** Specifies how the window title should be aligned across the **\*TOP/\*BOTTOM** of the window. The default for these parameters, if not specified, is **\*CENTER/\*TOP**. The default color is blue. The syntax of the parameter is (\*COLOR value).

This keyword is optional. Following is an example of the **WDWTITLE** keyword:

```
WDWTITLE ( (*TEXT TaxInformation') (*COLORWHT) *CENTER*TOP) '
```

## SDA

SDA currently does not support the WDWTITLE keyword. However, you can use SDA to design your window and then use DDS to add the WDWTITLE.

## Recommendation:

The window title, if used, should be normal text that is centered on the top line, without any special attributes or colors in order to avoid any leading attribute byte.

# WINDOW example: DDS

IBM i

```

-----+-----+-----+-----+-----+-----+-----+
000400  ** Pop-Up window Format
000500  A          R WINDOWFMT
000600  A
000700
000800  A
000900  A
001000  A
001100
001200  A
001300  A
001400  A
001500
001600  A          LINE      3S 0P
001700  A          COL       3S 0P
001800
001900  A
002000  A
002100  A
002200  A
002300
002400  ** Dummy Format to prevent display clearing
002500  A          R DUMMY
002600> A
002700  A          ASSUME
002800
-----+-----+-----+-----+-----+-----+-----+

```

© Copyright IBM Corporation 2009

Figure 2-20. WINDOW example: DDS

AS075.0

## Notes:

This visual shows a simple example of the use of the window keywords, **WINDOW**, **WDWTITLE**, and **WDWBORDER**.

This is an example of a pop-up window. It overlays the formats that are currently displayed on the screen.

Very often you want to design your pop-up window as a shared, stand-alone function to avoid duplication of code. This means the window is provided by a separate display file, processed by a different program. This introduces a minor complication.

When a new display file is opened, the system clears the existing display. The use of **OVERLAY**, and so forth has no effect. To avoid this clearance, we use the **ASSUME** (or **KEEP**) keyword at record-level. The record format that the **ASSUME** is associated with need not be displayed--hence, the **DUMMY** format in the example. (**DUMMY** must have at least one field defined.)

# WINDOW example: RPG IV

IBM i

```
100001  FPopUpWdw  CF   E          WorkStn
100002
100003  /Free
100004    Line = 10;
100005    Col  = 20;
100006
100007  ExFmt WindowFmt;
100008  *Inlr = *On;
100009  /End-Free
100010
```

© Copyright IBM Corporation 2009

Figure 2-21. WINDOW example: RPG IV

AS075.0

## Notes:

This program sets the coordinates for the window and displays it.

The program sets the line and column number for the window coordinates and overlays any other format on the screen without clearing it.

## WINDOW example: Result

IBM i

© Copyright IBM Corporation 2009

Figure 2-22. WINDOW example: Result

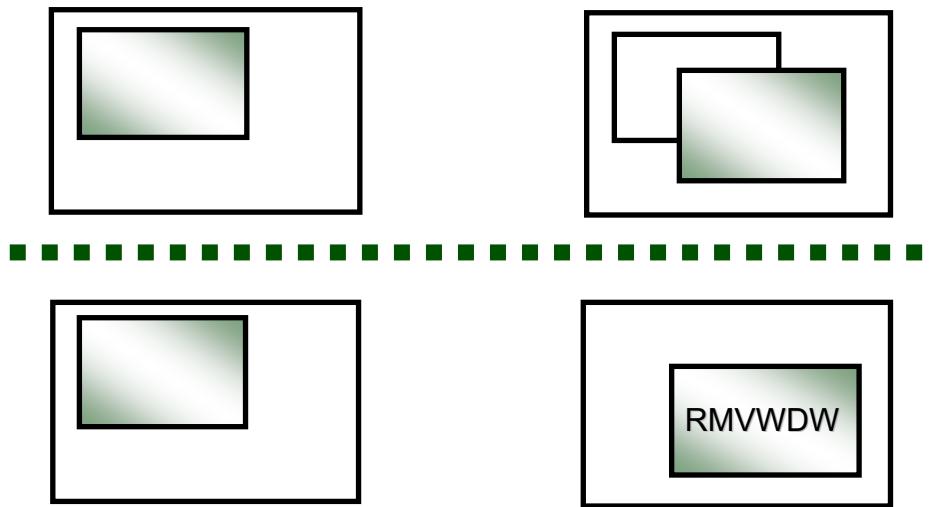
---

AS075.0

## **Notes:**

# RMVWDW keyword: Remove window

IBM i



© Copyright IBM Corporation 2009

Figure 2-23. RMVWDW keyword: Remove window

AS075.0

## Notes:

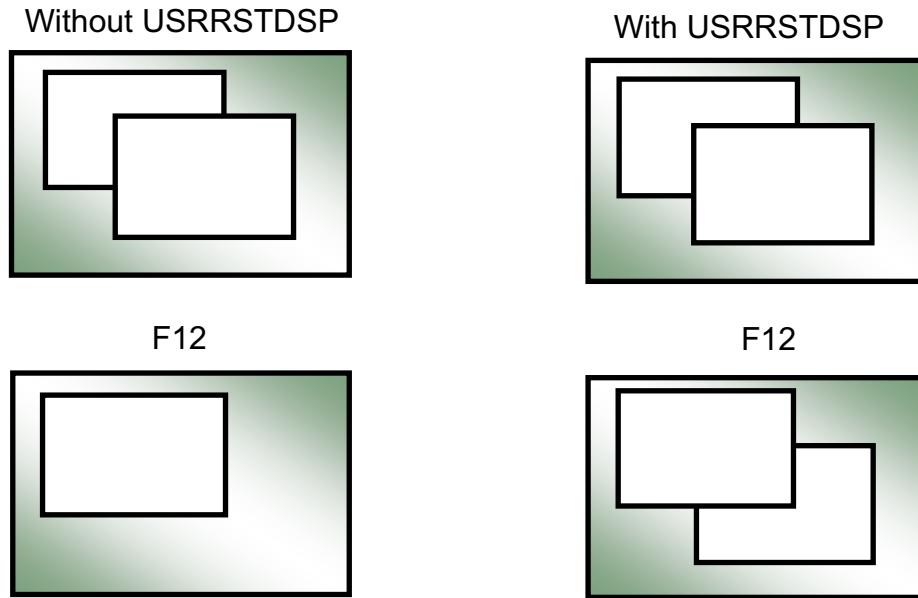
This keyword allows you to remove all currently displayed windows prior to displaying the window that has this keyword coded in it. RMVWDW must be coded within a window record format along with the WINDOW keyword.

The visual shows a possible use:

1. In the first example, the second window becomes active and the first window remains on the screen, displayed in the background. RMVWDW was *not* used.
2. In the second example, the second window becomes active and now the first window disappears. This happens because RMVWDW is coded in the DDS for the second window. It removes the first window from the display when the second window is displayed.

# USRRSTDSP keyword: User restore display

IBM i



© Copyright IBM Corporation 2009

Figure 2-24. USRRSTDSP keyword: User restore display

AS075.0

## Notes:

When this record-level keyword is specified on a window record layout, the system does *not* save underlying formats and restore them as needed. The program manages the display.

The left side of the visual shows what happens when **USRRSTDSP** is *not* coded with the second window format. The system manages the display by saving and restoring underlying windows. When F12 is pressed while window 2 is active (indication of being finished with window 2), the system restores window 1 automatically.

The right side of the visual shows system management of the same situation. In this case, **USRRSTDSP** is coded with window 2. When F12 is pressed while window 2 is active, the system responds with no restore of the underlying window 1. The program must restore the underlying display, possibly by rewriting window 1 to the display.

# USRRSTDSP user restore display: Example

IBM i

```

100039    ** Blue Window
100040    A      R BLUEFMT
100041    A 50
100042    A 51
100043    A
100044    A
100045    A
100046    A
100047    A
100048
100049    ** White Window
100050    A      R WHITEFMT
100051    A 50
100052    A 51
100053    A
100054    A
100055    A
100056    A
100057    A
100058
100059    ** Green Window
100060    A      R GREENFMT
100061    A 50
100062    A 51
100063    A
100064    A
100065    A
100066    A
100067    A

RMUWDW
USRRSTDSP
WDWTITLE((*TEXT 'Blue Window') +
(*COLOR BLU) *LEFT *TOP)
WINDOW(5 15 5 25)
WDWBORDER((*COLOR BLU) (*DSPATR RI)-
(*CHAR '12345678'))

RMUWDW
USRRSTDSP
WDWTITLE((*TEXT 'White Window') +
(*COLOR WHT) *LEFT *TOP)
WINDOW(7 25 5 25)
WDWBORDER((*COLOR WHT) (*DSPATR RI)-
(*CHAR '12345678'))

RMUWDW
USRRSTDSP
WDWTITLE((*TEXT 'Green Window') +
(*COLOR GRN) *LEFT *TOP)
WINDOW(9 35 5 25)
WDWBORDER((*COLOR GRN) (*DSPATR RI)-
(*CHAR '12345678'))

```

© Copyright IBM Corporation 2009

Figure 2-25. USRRSTDSP user restore display: Example

AS075.0

## Notes:

**USRRSTDSP** prevents the system from automatically saving and restoring the underlying display after windows that were displayed are removed. Bypassing these save and restore operations, where possible, speeds up your application. You can also use the **USRRSTDSP** keyword to make an earlier window in a series pop up and overlay later windows, and to make two windows seem active at the same time.

# Simple window: DDS

IBM i

```

000001 A** Pop-Up window Format
000002 A R WINDOWFMT
000003
000004 A
000005 I
000006 A
000007 A
000008
000009 A
000010 A
000011
000012 A
000013 A
000014 A
000015 A JOBNAME 10 0 2 16
000016 A USERID 10 0 3 16
000017 A JOBNUMBER 6 00 4 16
000018 A
000019 A
000020
000021 ** Dummy Format to prevent display clearing
000022 A R DUMMY
000023 A
000024 A ASSUME
000025 A 2 4' '

```

© Copyright IBM Corporation 2009

Figure 2-26. Simple window: DDS

AS075.0

## Notes:

This is another example of a very simple window that overlays the existing display with information that is extracted from the Program Status Data Structure.

# Simple window: RPG IV

IBM i

```
100001 F WdwDemo4 CF E          WorkStn
100002
100003 D                   SDS
100004 D JobName           244   253A
100005 D UserId            254   263A
100006 D JobNumber         264   269S 0
100007
100008 /Free
100009   ExFmt WindowFmt;
100010   *Inlr = *ON;
100011 /End-Free
```

© Copyright IBM Corporation 2009

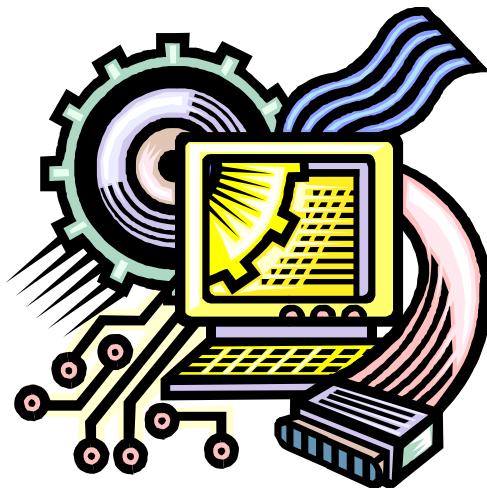
Figure 2-27. Simple window: RPG IV

AS075.0

## Notes:

# Machine exercise: Using DDS windows

IBM i



© Copyright IBM Corporation 2009

Figure 2-28. Machine exercise: Using DDS windows

AS075.0

## Notes:

Perform the machine exercise “Using DDS windows.”

## Unit summary

Having completed this unit, you should be able to:

- Create display file DDS that supports OVERLAY, PUTOVR, OVRATR, and OVRDTA keywords
- Explain the purpose of OVERLAY, PUTOVR, OVRATR, and OVRDTA keywords
- Code RPG IV interactive programs that take advantage of display files that contain display files with the OVERLAY, PUTOVR, OVRATR, and OVRDTA keywords
- Design windows using DDS and write programs to use windows

© Copyright IBM Corporation 2009

Figure 2-29. Unit summary

AS075.0

### Notes:

This unit has given you a solid foundation for adding more function and flexibility to your interactive applications. However, there are other things that you can try on the job:

- Implement subfiles in DDS windows. We discuss subfiles at length in a later unit but we will not discuss DDS windows in subfile applications.
- Implement Menu Bars using DDS.
- Build Help displays using the User Interface Manager (UIM).
- Develop GUI interface using CGI programs, WebSphere and WebFacing. AS10/S6199 introduces you to some of these techniques.

An excellent reference for some of advanced techniques is the ITSO Redbook *Who Knew You Could Do That with RPG IV?* Many of these techniques are discussed in forums such as the **Midrange.com RPG400-L**.

# Unit 3. Using arrays

## What this unit is about

This unit describes how to define and use arrays in RPG IV. It also discusses the RPG tables briefly.

## What you should be able to do

After completing this unit, you should be able to:

- Define an array in an RPG IV program
- Use the %LookUpxx and %Xfoot BIFs
- Define a table in an RPG IV program

## How you will check your progress

- Machine exercise
  - To an existing program, the students add RPG IV code to define and process arrays.
- During this exercise, the students add RPG IV code to:
  - Define a compile-time array
  - Store and access data in array elements
  - Access array elements using an index

# Unit objectives

IBM i

After completing this unit, you should be able to:

- Define an array in an RPG IV program
- Use the %LookUpxx and %Xfoot BIFs
- Define a table in an RPG IV program

---

© Copyright IBM Corporation 2009

Figure 3-1. Unit objectives

AS075.0

## **Notes:**

# What are arrays and tables?

IBM i

- Groups of related data:
  - All elements same data type
  - Operations on data in common fashion
  - One dimensional

- 
- Arrays:
    - Access using index
    - Sequential or random access
  - Tables:
    - No index
    - Sequential access only

© Copyright IBM Corporation 2009

Figure 3-2. What are arrays and tables?

AS075.0

## Notes:

Tables and arrays have been around for many years. Although similar to arrays, tables are not as flexible. The only reason we will mention tables in this class is that you might encounter them in application code that needs to be maintained.

A table or array contains data of the same type and definition.

Some sample uses for arrays are:

- Group inventory quantities for items that are located in many different locations. You can maintain separate quantities but can do cumulative operations simply. You can access any element directly by using the index and pointing to that element.
- Allow easy crossfooting of sales or cost history for a time period.

Some sample uses for tables are:

- Relate location codes to location descriptions
- Associate income amounts with tax percentages

Generally, tables are most often used to relate groups of data on an element-by-element basis. Tables are searched sequentially element by element.

Arrays are widely used in commercial applications. Operations and BIFs can be used to manipulate the data elements of arrays individually or as a group (**%XFOOT**, **%LOOKUPxx**). We describe how to use them with arrays and tables.

Arrays provide the functions of tables and more. The emphasis of this class is on arrays for this reason. Where appropriate, we discuss the differences between these two types of grouped data.

# What is an array?

IBM i

- Assume we have a set of 12 fields, named SALES.
- Each individual field holds sales for a specific month.
- Each field is numeric, eight digits, two decimals.

<u>Month</u>	<u>Total Sales by Month</u>	
January	125876.15	(1)st element
February	78421.12	(2)nd element
March	2248334.87	(3)rd element
April	245453.15	(4)th element
May	312585.35	(5)th element
June	299542.76	(6)th element
July	342664.01	(7)th element
August	376012.00	(8)th element
September	357886.12	(9)th element
October	423998.67	(10)th element
November	422554.23	(11)th element
December	456334.12	(12)th element

© Copyright IBM Corporation 2009

Figure 3-3. What is an array?

AS075.0

## Notes:

An *array* is a *contiguous set of data elements* that are the same length and data type. Each *element* can be accessed using a numeric value, called an **index**, combined with the array name. You can address a unique element or the complete array.

If the name of the set of data in the visual was **SALES**, we could work with the data for June by addressing **SALES(6)**. The **6** is the index to the **SALES** array. The index can be a numeric literal or a variable. You can write a single line of code and loop through varying values of the index rather than process a number of sequential lines of code that are exactly the same except for the address of the variable.

# Processing data in an array

IBM i

- Field by field:

```

000001 D TotSales      S          +2   LIKE(JanSales)
000002
000003 /Free
000004   TotSales = JanSales
000005     + FebSales
000006     + MarSales
000007     + AprSales
000008     + MaySales
000009     + JunSales
000010     + JulSales
000011     + AugSales
000012     + SepSales
000013     + OctSales
000014     + NovSales
000015     + DecSales;
000016
000016 /End-free
IS 07V2LIB/QRPGLESRC(FIG34A) resequenced and saved

```

- Using an array:

Row	Column 1	Insert
000001	D TotSales	S +2 LIKE(JanSales)
000002	D Sales	S 8S 2 DIM(12)
000003	D Mon	S 2 0 INZ(1)
000004		
000005	/Free	
000006	For Mon = 1 to 12;	
000007	TotSales = TotSales + Sales(Mon);	
000008	EndFor;	
000009	/End-free	

© Copyright IBM Corporation 2009

Figure 3-4. Processing data in an array

AS075.0

## Notes:

Using an array can mean fewer lines of code. In this case, we assume that the **JanSales** through **DecSales** fields are defined in a database file. Using **Mon** as the index to the array (where **Mon** varies from **1** to **12**), we add the value of each element of the array **SALES**.

Our alternative, and we exaggerate in the example, is to code an expression to add each and every **mmmSALES** field into the accumulator **TotSales**.

# Defining an array

IBM i

```
D Sales      S          8S 2 DIM(12) <===== 1
D Mon       S          2S 0
```

- Arrays are defined on the D-Spec.
- The DIM(ension) keyword is used to indicate an array.
- There are **12** elements in this array.

© Copyright IBM Corporation 2009

Figure 3-5. Defining an array

AS075.0

## Notes:

Here we have defined a basic array that holds our sales figures for the past twelve months:

- In DDS, we cannot describe an array as multiple individual fields. Our 12 months of sales data might be defined as a single field of 60 digits (stored in 60 bytes as zoned data).
- On the D-spec, (1), **Sales** is described as an array of **12** elements, each **8 digits** in length.

We can now do various operations on the array, such as:

- Crossfoot (calculate the sum of the set) all the elements and obtain a 12-month total sales. We could use the **%xfoot** BIF.
- Code a Do/For loop to find the month with the highest sales.

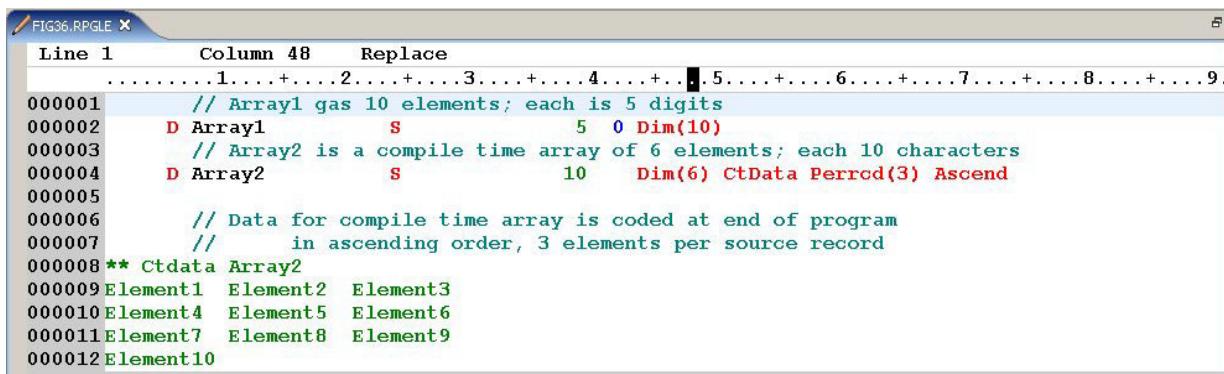
Arrays provide a great deal of convenience compared to trying to operate on each field individually. The Sales array is built from a externally described file. In DDS, the 12 fields are contiguous and can be defined as one large field.

We can move the contents of the sales history data from the sales record into the Sales array. Then we can process the data in element form. In the next unit, we discuss data structures and how they can be used to group and redefine data.

# Summary of D-Spec keywords for arrays

IBM i

Keyword	Description
DIM(number)	Number of elements in array
ASCEND/DESCEND	Order of elements in array
PERRCD(number)	# elements for compile time table/array
FROMFILE/TOFILE	From/To file for pre-run time array/table
EXTFMT	External data type for compile and pre-run time arrays
ALT(array_name)	Compile/preruntime array alternating format
CTDATA	Indicates compile time data



```

FIG36.RPGLE x
Line 1      Column 48   Replace
.....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8....+....9.
000001 // Array1 has 10 elements; each is 5 digits
000002 D Array1      S          5 0 Dim(10)
000003 // Array2 is a compile time array of 6 elements; each 10 characters
000004 D Array2      S          10    Dim(6) Ctdata Perrcd(3) Ascend
000005
000006 // Data for compile time array is coded at end of program
000007 // in ascending order, 3 elements per source record
000008 ** Ctdata Array2
000009 Element1 Element2 Element3
000010 Element4 Element5 Element6
000011 Element7 Element8 Element9
000012 Element10

```

© Copyright IBM Corporation 2009

Figure 3-6. Summary of D-Spec keywords for arrays

AS075.0

## Notes:

Arrays and tables are defined on the D-spec. This visual highlights the keywords associated with the definition of arrays.

In this visual, note the following points:

- The DIM keyword identifies the standalone fields **Array1** and **Array2** as arrays. Note that both are one-dimensional arrays. You might be familiar with the term *vector*. These are vectors.
- The maximum number of elements or the total size of an array allowed is 16,773,104. The total size of the array (all of its elements) cannot exceed 16,773,104 bytes. This is a system limitation established at V6.1 of IBM i.
- Arrays can be loaded with data in three different ways:
  - Compile time:** The data is included with the source program. The **CTDATA** keyword is used to specify that data is included with the source program.
  - Pre-execution time:** The data is loaded from a file that is to be completely read at program initialization.

- **Execution time:** The data is loaded from records in a data file as requested by the program. This is the most common way of loading data.
- **Array2** is loaded at compile time. Notice the **CTDATA** keyword. Also, notice it again at the end of the source program. The data for the array follows the **\*\* CTDATA Array2** statement. This identifies the data as belonging to **Array2**. The **PERRCD** keyword is used to specify how many array elements are included per record (in this case, source record).

The type of array that you use depends on the volatility of your data. If the data is fixed and never changes, you may use compile time or preexecution arrays.

Notice that if the data in a compile time array ever changes, you have to update the array using SEU or CODE and then recompile your program. So, don't place information such as sales tax or income tax information in a compile time table or array!

For your reference, other keywords are documented below:

- **ASCEND** or **DESCEND**: To specify that the array is sequenced.
- **TOFILE**: If you want to write out the array to a file at EOJ, specify TOFILE. The data will be written at LR time.
- **FROMFILE**: The name of the file that contains the data for the preexecution array (or table).
- **EXTFMT**: Used to define the format of the data in the data record (for example, B=binary, S=signed numeric, I=integer).
- **ALT**: Alternating sequence is to be used.

Finally, note that for compile time arrays (or tables), at the end of your RPG source you will code one **\*\*CTDATA** statement for each table and array followed by its data. It is not mandatory to include the array or table name beside the **\*\*CTDATA** but it is good practice. If you use **\*\*CTDATA array/table\_name**, you must do so consistently for the program. If you do not specify the array/table\_name, the compile time tables and arrays are loaded with data based on the order in which they are defined on the D specification.

You must not mix these methods. The compiler does not allow it.

# Define, load, and access array data

IBM i

The image shows two windows from the IBM i interface. The top window, titled 'FIG37A.PF', displays a physical file definition (D-Spec) for 'CUSTMASTER'. It includes fields like CUSTNUM (5 0), CUSTNAME (25), JANSALES (8 2), FEBSALES (8 2), and MARSALES (8 2). The bottom window, titled 'FIG37B.RPGLE', shows an RPGLE program. The code defines an array 'Sales' (8S 2 Dim(12)), initializes it with data from 'CUSTMASTER' (JANSALES, FEBSALES, MARSALES), calculates a total sales ('TotSales'), and then reads the file again. The code is as follows:

```

Line 22    Column 1      Insert
000001      FCustMast IF E           Disk
000002      // Define array
000003      D Sales      S           8S 2 Dim(12)
000004      // Define array index
000005      D Mon        S           2S 0
000006      D TotSales   S           +2   Like(Sales)
000007
000008      /Free
000009      Read CustMast;
000010      Dow Not %Eof(CustMast);
000011          Sales(1) = JanSales;      // For each record, load array
000012          Sales(2) = FebSales;
000013          Sales(3) = MarSales;
000014          :
000015      FOR Mon = 1 to 12;
000016          TotSales = Sales(Mon) + TotSales; // For each record, calculate
000017          // TotSales
000018      EndFor;
000019
000020      Read CustMast;
000021      EndDo;
000022      :
000023      /End-free

```

© Copyright IBM Corporation 2009

Figure 3-7. Define, load and access array data

AS075.0

## Notes:

This example illustrates the steps that you must use to successfully implement an array in a program:

- **Define** the array on the D-Spec.
- **Load** the array with data (from a physical file).
- **Access** the array using an index.

In this case, we have a file that has sales data collected on a monthly basis for each of our customers. We want to calculate a simple year to date sales total.

First, we must define the array. What tells the compiler that **Sales** is an array?

Next, we load the array with data. In this case, as we read the file **CustMast**. The array is loaded with data each time a record is read from the file.

Finally, we access the array using an index, **MON**. To access an array element, we specify the array name and its index surrounded by parentheses. When you name your array, remember that you will access individual elements using an index. Allow for the size of the

index field name and the parentheses that must surround the index field in addition to the array name. Free-format calculations and extended names enable you to use a meaningful array name with an index.

You must follow several rules when using an array index:

- The index can be an RPG numeric variable or a literal numeric.
- The value of the index must not be negative, zero, or greater than the number of elements in the array. If it is, you get an execution error.

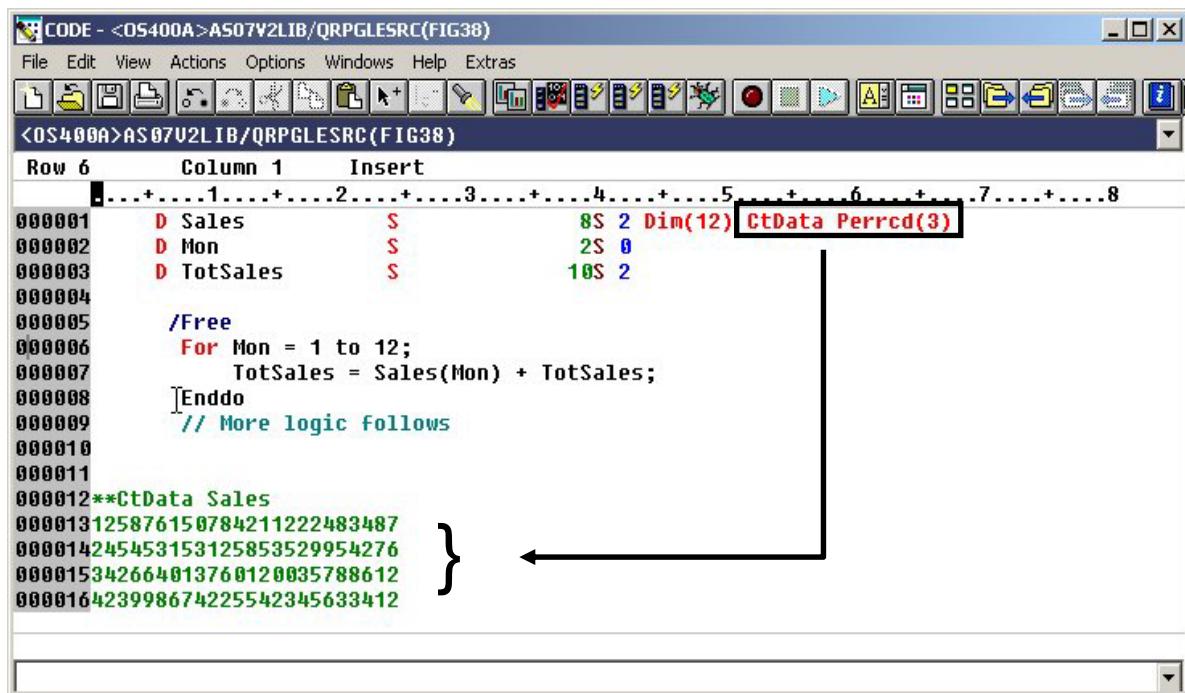
Arrays (and tables) can be loaded at run time as illustrated in this visual. You can also specify that a table can be initialized with data at compile time, using the CTDATA keyword. A compile-time array contains data in the source program. An example is shown in a subsequent visual. Finally, data can be loaded into an array at preexecution time. A pre-execution time array is loaded from a file during program initialization.

Where would you use each method of initialization?

- **Compile time:** For data that rarely changes (if ever) such as days of the week, months, and so on.
- **Pre-execution time:** For data that changes occasionally but does not vary throughout the execution of the program or the array only contains a single set of values. An example might be tax tables that usually change on an annual basis.
- **Execution time:** For data that is volatile or that changes with each record read. An example might be a set of sales for each customer by month.

# Compile time load

IBM i



The screenshot shows the IBM i Code editor interface. The title bar reads "CODE - <OS400A>AS07V2LIB/QRPGLESRC(FIG38)". The menu bar includes File, Edit, View, Actions, Options, Windows, Help, and Extras. The toolbar contains various icons for file operations and code navigation. The code editor window displays a program named "AS07V2LIB/QRPGLESRC(FIG38)". The code uses Row and Column numbering. A red box highlights the line "8S 2 Dim(12) CtData Perrcd(3)" in row 1. A large black bracket on the right side of the code area groups the entire section from row 12 down to row 16. The code itself is as follows:

```

Row 6      Column 1      Insert
000001    D Sales      S          8S 2 Dim(12) CtData Perrcd(3)
000002    D Mon        S          2S 0
000003    D TotSales   S          10S 2
000004
000005    /Free
000006      For Mon = 1 to 12;
000007          TotSales = Sales(Mon) + TotSales;
000008      Enddo
000009      // More logic Follows
000010
000011
000012 **CtData Sales
000013 125876150784211222483487
000014 245453153125853529954276
000015 342664013760120035788612
000016 423998674225542345633412
}

```

© Copyright IBM Corporation 2009

Figure 3-8. Compile time load

AS075.0

## Notes:

A compile-time array is specified on the D-spec plus the keyword **CTDATA**. In addition, on a definition specification, you can specify the number of array entries in an input record using the **PERRCD** keyword. If the keyword is not specified, the number of entries per record defaults to 1.

# When to use arrays

## Sales History Record

### Database

**CustNum**  
**CustName**  
**JanSales**  
**FebSales**  
**MarSales**  
:  
:  
**OctSales**  
**NovSales**  
**DecSales**

### RPG IV Program

**CustNum**  
**CustName**  
**Sales(Mon) Mon = 1 to 12**

© Copyright IBM Corporation 2009

Figure 3-9. When to use arrays

AS075.0

### **Notes:**

When deciding if an array is the best way to organize and access data, consider the following points:

- Is the data grouped in single record formats? If it is spread across multiple records, an array probably won't fit the need.
- In the data group, is each data field (or element) defined exactly the same? All elements of an array are of the same data type and length.
- Do you want to access the data as individual fields and as a group?
- How is the data defined in the database? Often, each element is defined as a unique field in DDS because you want to access the data using i5(iSeries) tools, such as QUERY.

Because the database cannot define an array, the program must contain the code needed to move the data from the DB fields into the corresponding array elements.

# Naming arrays and tables

IBM i

## ARRAYS

**Name can be any valid RPG name**

**Index used to address elements**

**Index can be literal / variable**

**SALES(MON)**

**Compile, pre-run or execution fill**

**Search with %LOOKUPxx**

## TABLES

**Name *must* begin with TAB**

**No index facility**

**TABCOD**

**Compile or pre-run fill only**

**Search with%TLOOKUP**

© Copyright IBM Corporation 2009

Figure 3-10. Naming arrays and tables

AS075.0

### **Notes:**

Arrays can be named anything you want provided that it is a valid RPG IV name. When you consider a name for the array, remember that you reference individual elements and need to allow space for the parentheses and the index variable name. Some examples are:

- **Sales(Mon):** An array of 12 elements (**Mon = 1 to 12**). Remember that the index *cannot be greater than 12. Nor can it be zero or negative* when you are processing the Sales array. If the index is zero or negative, you get an execution time error and control is passed to the RPG IV Default Error Handler.
- **QtyAv(Loc):** Inventory quantity available for shipment from nine warehouse locations. **Loc** is a numeric variable varying from 1 to 9.

Table names can also be any valid RPG IV name. But, they must begin with the characters **TAB**. This is how the compiler recognizes a table versus an array. The **DIM** keyword is still used to specify the number of elements in the table. The other difference between tables and arrays is that you cannot index a table. When you process a table, you must search it sequentially using **%TLookUpxx** BIFs.

# Moving array data (1 of 2)

IBM i

```

0001  FSalesFile IF   E          K Disk
0002
0003  D Sales      S          8S 2 DIM(12)
0004  D AnnSales   S          9S 2 DIM(5)
0005  D HistSales  S          9S 2 DIM(5)
0006  D Mon        S          2  0
0007  D Yr         S          1  0 INZ(1)
0008
0009  /Free
0010 // Accumulate total sales for past five years
0011 For Year = 1 to 5;
0012
0013     Read S1sRec; // Read a record
0014     IF NOT %eof(SalesFile);
0015         ExSR LoadArrays;
0016 // The Sales array is in the S1sRec
0017 // Sums all array elements
0018     AnnSales(Yr) = %xfoot(Sales);
0019     EndIF;
0020 EndFor;
0021 // Move the AnnSales array in one operation
0022 HistSales = AnnSales;
0023 BegSR LoadArrays;
0024 // Logic
0025 EndSR LoadArrays;
0026 *inlr = *on;

```

© Copyright IBM Corporation 2009

Figure 3-11. Moving array data (1 of 2)

AS075.0

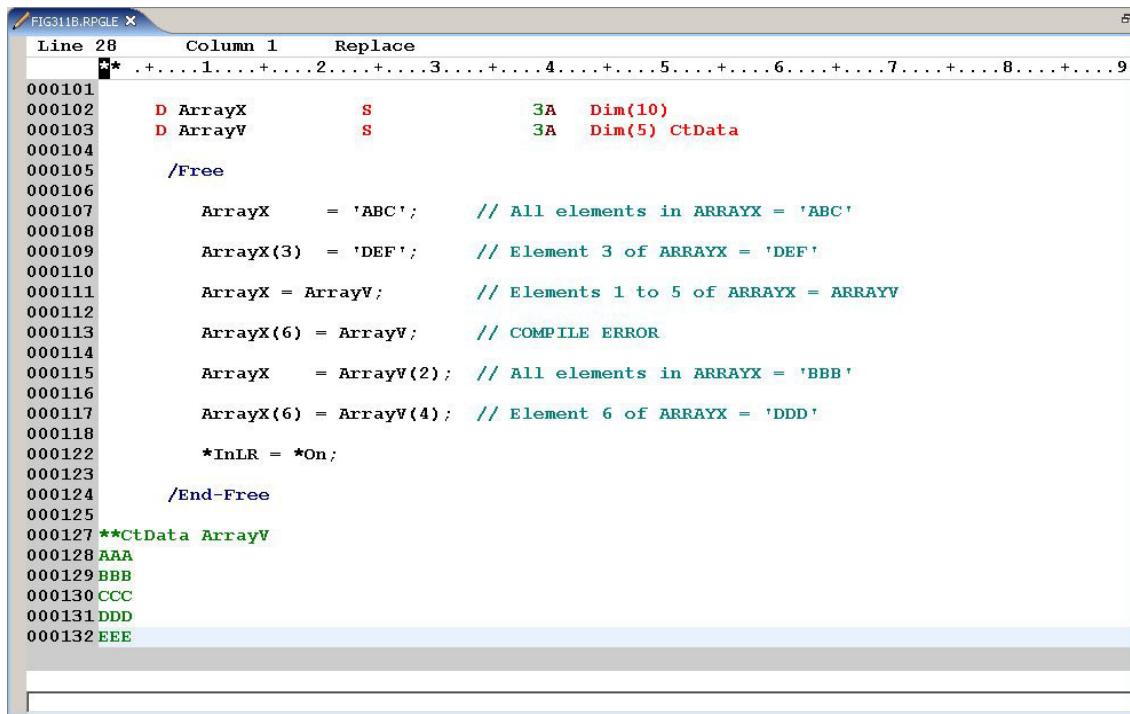
## Notes:

Use Eval (in this case implicitly) to move data from one array to another. Notice that all elements are moved in a single statement:

- As usual, the data types must match on both sides of the Eval.
- You may have different lengths on both sides of the Eval but be careful of truncation of character data or numeric overflow.
- You may also have arrays of different sizes on both sides of the equation.
- You may also use the EvalR opcode to right adjust character data.
- The elements are operated on one by one until either of the arrays runs out of elements.

## Moving array data (2 of 2)

IBM i



```

FIG311B.RPGLE x
Line 28      Column 1      Replace
*+ .+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8....+....9.
000101      D ArrayX          S           3A   Dim(10)
000102      D ArrayV          S           3A   Dim(5) CtData
000103
000104
000105      /Free
000106
000107      ArrayX      = 'ABC';      // All elements in ARRAYX = 'ABC'
000108
000109      ArrayX(3) = 'DEF';      // Element 3 of ARRAYX = 'DEF'
000110
000111      ArrayX = ArrayV;      // Elements 1 to 5 of ARRAYX = ARRAYV
000112
000113      ArrayX(6) = ArrayV;      // COMPILE ERROR
000114
000115      ArrayX      = ArrayV(2); // All elements in ARRAYX = 'BBB'
000116
000117      ArrayX(6) = ArrayV(4); // Element 6 of ARRAYX = 'DDD'
000118
000119      *InLR = *On;
000120
000121      /End-Free
000122
000123
000124
000125
000126
000127 **CtData ArrayV
000128 AAA
000129 BBB
000130 CCC
000131 DDD
000132 EEE

```

© Copyright IBM Corporation 2009

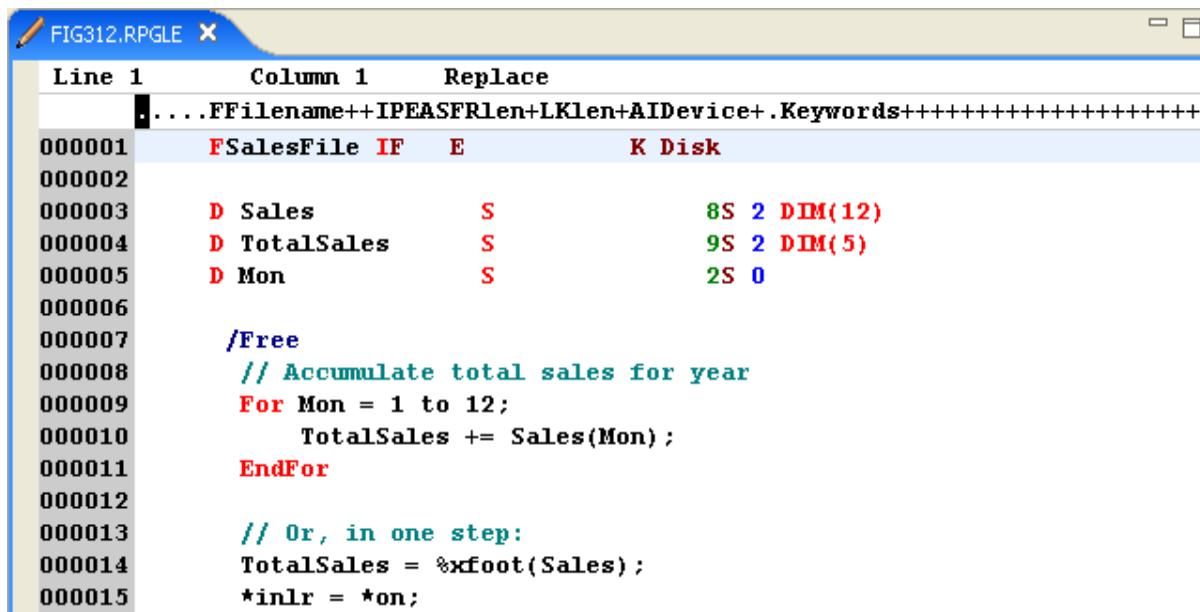
Figure 3-12. Moving array data (2 of 2)

AS075.0

### Notes:

# Crossfooting arrays

IBM i



The screenshot shows an IBM i terminal window with the title 'FIG312.RPGLE'. The window displays RPGLE code for crossfooting arrays. The code defines a file 'FSalesFile' with fields Sales (8S 2 DIM(12)), TotalSales (9S 2 DIM(5)), and Mon (2S 0). It then uses a free form section to accumulate total sales for the year by summing the monthly sales. Finally, it uses the %Xfoot function to sum all elements of the array into the TotalSales field.

```

Line 1      Column 1      Replace
.....FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords+++++
000001      FSalesFile IF   E          K Disk
000002
000003      D Sales        S          8S 2 DIM(12)
000004      D TotalSales    S          9S 2 DIM(5)
000005      D Mon          S          2S 0
000006
000007      /Free
000008      // Accumulate total sales for year
000009      For Mon = 1 to 12;
000010          TotalSales += Sales(Mon);
000011      EndFor
000012
000013      // Or, in one step:
000014      TotalSales = %Xfoot(Sales);
000015      *inlr = *on;

```

© Copyright IBM Corporation 2009

Figure 3-13. Crossfooting arrays

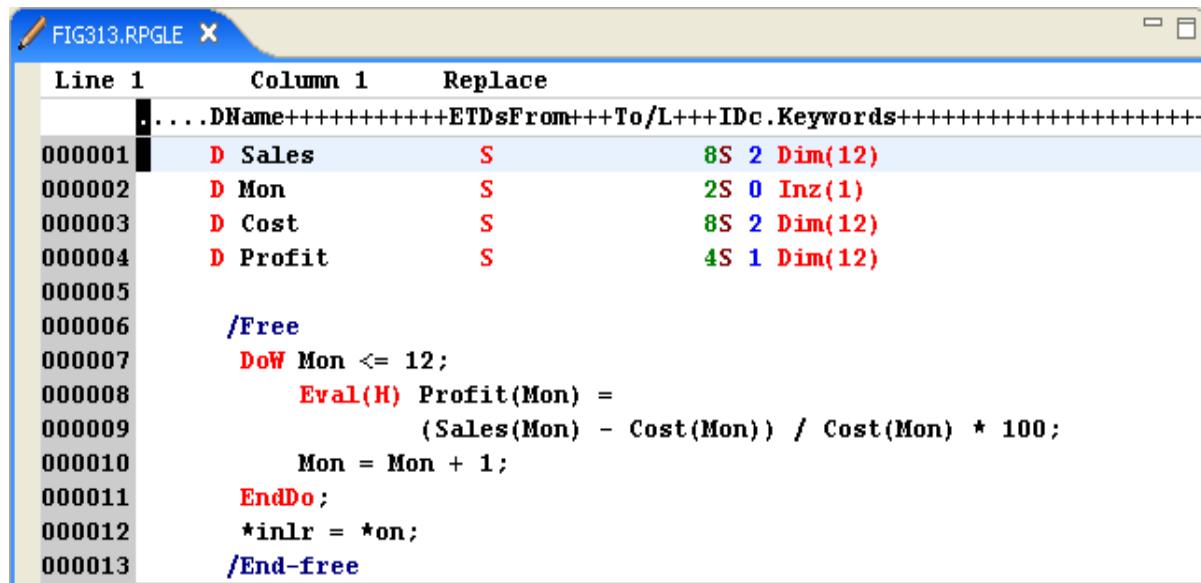
AS075.0

## Notes:

**%Xfoot** sums all the elements of an array together and places the sum into the accumulator field specified in the result field.

# Using arrays in a loop

IBM i



```

FIG313.RPGLE X

Line 1      Column 1      Replace
.....DName++++++ETDsFrom++To/L+++IDc.Keywords+++++
000001      D Sales      S          8S 2 Dim(12)
000002      D Mon       S          2S 0 Inz(1)
000003      D Cost       S          8S 2 Dim(12)
000004      D Profit     S          4S 1 Dim(12)
000005
000006      /Free
000007      DoW Mon <= 12;
000008          Eval(H) Profit(Mon) =
000009              (Sales(Mon) - Cost(Mon)) / Cost(Mon) * 100;
000010          Mon = Mon + 1;
000011      EndDo;
000012      *inlr = *on;
000013      /End-free

```

© Copyright IBM Corporation 2009

Figure 3-14. Using arrays in a loop

AS075.0

## Notes:

Notice how simple it is to operate on arrays. With a few lines of code, we can determine the profit by month for the past twelve months. This is a very simple example but imagine the coding that would be necessary without arrays.

Notice the half adjust operation extender (H).

# %LookUpxx

**%LOOKUPxx(arg:array{:startindex})**

**%LOOKUP:** Searches for an exact match

**%LOOKUPLE:** Searches for either an exact match, or the highest item lower than *arg*

**%LOOKUPLT:** Searches for the closest item lower than *arg*

**%LOOKUPGE:** Searches for either an exact match, or the lowest item higher than *arg*

**%LOOKUPGT:** Searches for the closest item greater than *arg*

- Successful match returns index value; 0 = unsuccessful.
- *arg* and array can be of any type, but they must match.
- *startindex* must be a non-float numeric with 0 decimals.
- For %LOOKUPxx, the array must be defined with ASCEND or DESCEND.

© Copyright IBM Corporation 2009

Figure 3-15. %LookUpxx

AS075.0

## Notes:

This visual shows the features of %LookUPxx for arrays. The rules and features for the table lookup BIFs, %TLookUpxx, are the same.

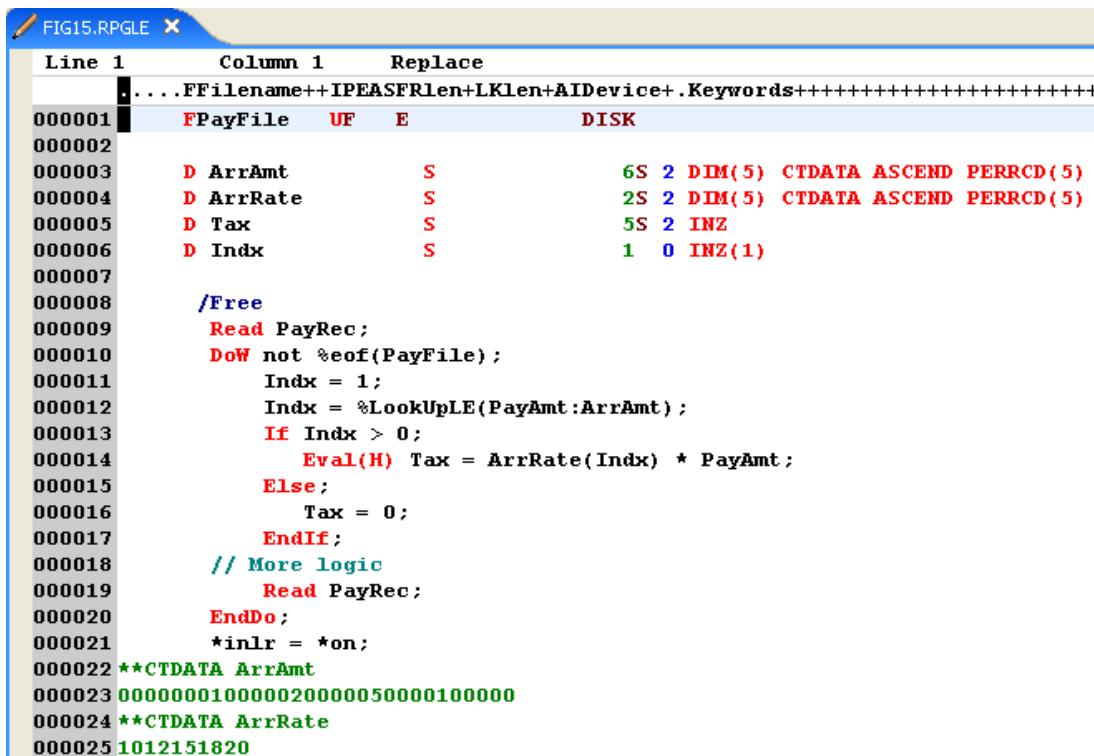
There is a fourth parameter that you can use that tells the BIF at which element to stop searching.

`%LOOKUP(arg :array {:startindex {:numelems}})`

Using the **%LookUPLT**, **%LookUPGE** or **%LookUPGT** requires that the array be sorted. We look at how to do this soon.

# %LookUpxx example

IBM i



```

FIG15.RPGLE X
Line 1      Column 1      Replace
. ....FFilename++IPEASRlen+LKlen+AIdevice+.Keywords+++++
000001    FPayFile  UF   E           DISK
000002
000003    D ArrAmt       S           6S 2 DIM(5) CTDATA ASCEND PERRCD(5)
000004    D ArrRate      S           2S 2 DIM(5) CTDATA ASCEND PERRCD(5)
000005    D Tax          S           5S 2 INZ
000006    D Indx         S           1  0 INZ(1)
000007
000008    /Free
000009        Read PayRec;
000010        DoW not %eof(PayFile);
000011            Indx = 1;
000012            Indx = %LookUpLE(PayAmt :ArrAmt);
000013            If Indx > 0;
000014                Eval(H) Tax = ArrRate(Indx) * PayAmt;
000015            Else;
000016                Tax = 0;
000017            EndIf;
000018            // More logic
000019            Read PayRec;
000020        EndDo;
000021        *inlr = *on;
000022    **CTDATA ArrAmt
000023 000000010000020000050000100000
000024    **CTDATA ArrRate
000025 1012151820

```

© Copyright IBM Corporation 2009

Figure 3-16. %LookUpxx example

AS075.0

## Notes:

This is an array application. Suppose we have an application where we must determine the rate of tax based upon an amount. The greater the amount, the higher the tax rate.

The **%Lookupxx** BIF searches the array based on the search argument, which is the first parameter. The search argument is matched to the array in the second parameter. The type of search depends on the suffix of the **%Lookupxx** BIF. In this case, we have the array (**ArrAmt**) organized such that we want to match the array element that is less than or equal to the search argument.

In this example, we want to search the array sequentially, element by element, until we find the element of the array **ArrAmt** that is equal to or lower than the value of **PayAmt**.

When **%LookULE** has a match (**Indx > 0**), we know that the corresponding element of **ArrRate** has the tax rate that we must apply. You reference the array element in **ArrRate** by using the array name, **ArrRate(Indx)**, in subsequent code as desired. The value of the **Indx** field that points to **ArrAmt** and **ArrRate** remains the same until the next search using **%LookUpLE** is performed.

Here are the elements of the two arrays, **ArrAmt** and **ArrRate** and the corresponding values of **PayAmt**:

PayAmt	ArrAmt	ArrRate
0.00 - 99.99	0	0.10
100.00 - 199.99	100	0.12
200.00 - 499.99	200	0.15
500.00 - 999.99	500	0.18
1000.00 - 9999.99	1000	0.20

This is an example of the way in which the %LookUPxx BIF can be used with an array. For several values of **PayAmt**:

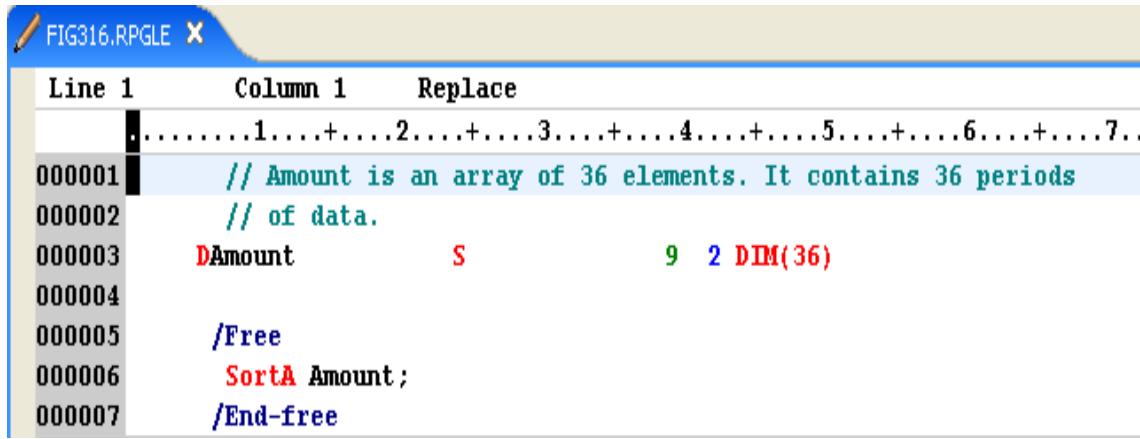
- If **PayAmt** = 150.00, then **ArrAmt** points to the second element and we will use **ArrRate(2)**, that is, 0.12
- If **PayAmt** = 997.00, then **ArrAmt** points to the fourth element and we will use **ArrRate(4)**, that is, 0.18

Notice that the two arrays were loaded at compile time (see the CTDATA keyword). This means of loading data works well if the data is very stable in nature (such as day number / day of the week or month number / month of the year). For a tax application, we would normally load the data from a file at pre-execution time.

Not shown is the corresponding **%TLookUpxx** BIF for use with tables (see the RPG IV Reference for further information).

# SortA

IBM i



The screenshot shows an RPGLE editor window titled "FIG316.RPGL". The code is as follows:

```

Line 1      Column 1      Replace
.....1....+....2....+....3....+....4....+....5....+....6....+....7...
000001      // Amount is an array of 36 elements. It contains 36 periods
000002          // of data.
000003      DAmount           S             9  2 DIM(36)
000004
000005      /Free
000006      SortA Amount;
000007      /End-free

```

© Copyright IBM Corporation 2009

Figure 3-17. SortA

AS075.0

### Notes:

SortA is an opcode that can be used to sequence your array. If sequence was specifically defined on the D-spec, the elements are sorted into that sequence. If no sequence was specified, then ascending is assumed.

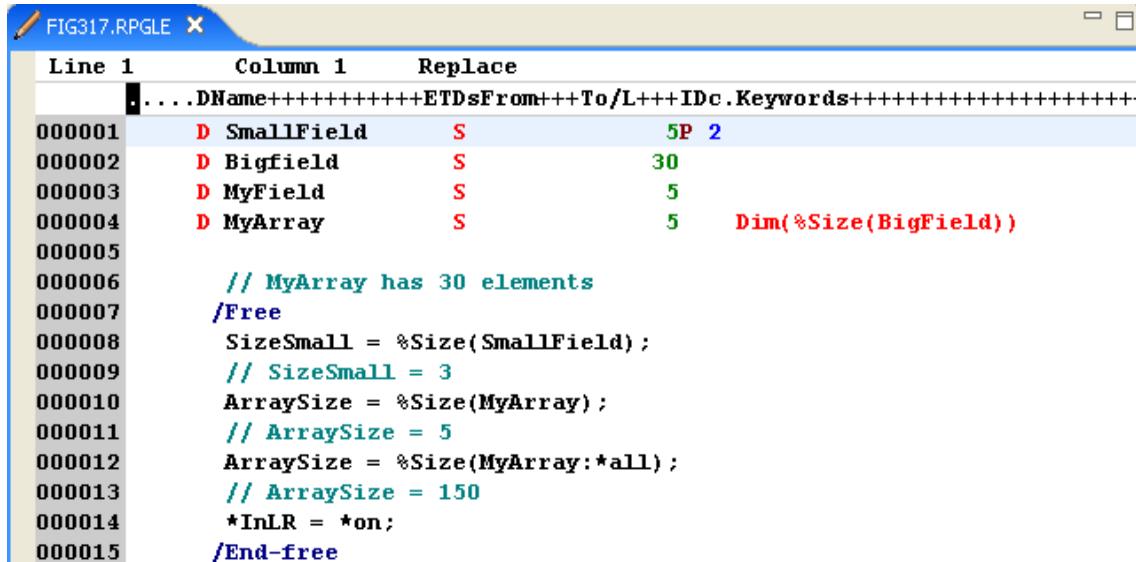
In the case in the visual, the SortA would produce an array of amounts sorted from smallest to largest. Ascending sequence is assumed because we did not specify any sequence on the D-spec.

## BIFs and arrays: %SIZE

IBM i

`%SIZE(name{::*ALL})`

- Returns storage size (in bytes) occupied by the named item



The screenshot shows the IBM i RPGLE editor window titled 'FIG317.RPGLE'. The code in the editor is as follows:

```

Line 1      Column 1      Replace
.....DName++++++ETDsFrom+++To/L+++IDc.Keywords+++++
000001      D SmallField    S          5P 2
000002      D Bigfield     S          30
000003      D MyField      S          5
000004      D MyArray       S          5      Dim(%Size(BigField))
000005
000006      // MyArray has 30 elements
000007      /Free
000008      SizeSmall = %Size(SmallField);
000009      // SizeSmall = 3
000010      ArraySize = %Size(MyArray);
000011      // ArraySize = 5
000012      ArraySize = %Size(MyArray:.*all);
000013      // ArraySize = 150
000014      *InLR = *on;
000015      /End-free

```

© Copyright IBM Corporation 2009

Figure 3-18. BIFs and arrays: %SIZE

AS075.0

### Notes:

As you can see from this example, built-in functions can be used in calculations as well as on the D-spec.

When used on the D-spec, the value of the **%SIZE** built-in function is determined at compile time and the argument cannot be an expression.

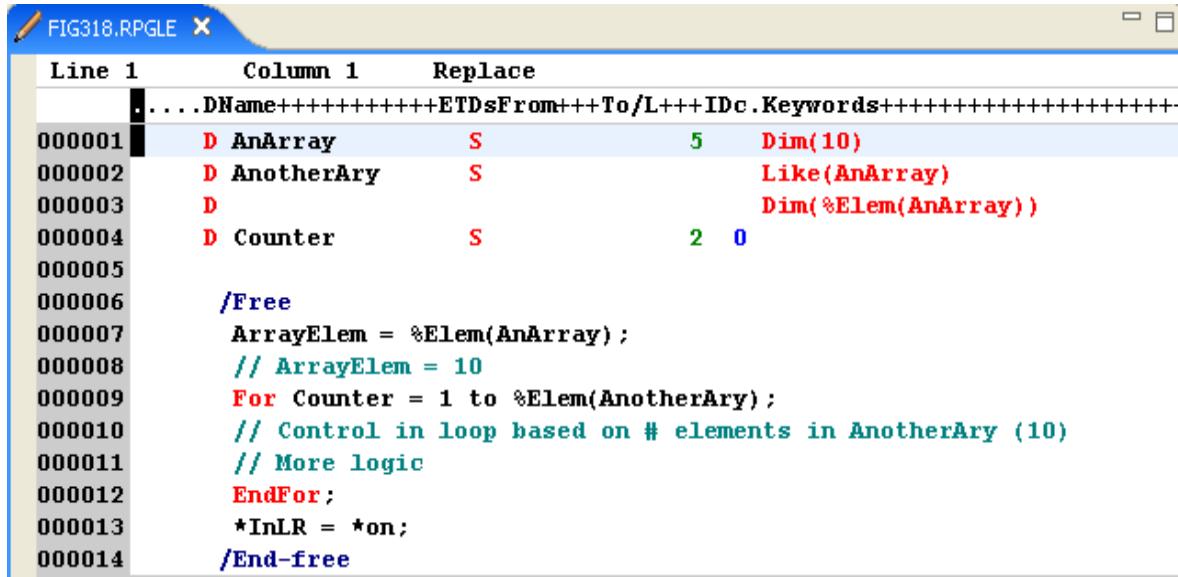
The **\*ALL** option of the **%Size** BIF is used with arrays. If **\*ALL** is specified as the second parameter for **%SIZE**, the value returned is the storage taken up by all elements or occurrences. If you are working with an array and do not specify **\*ALL**, the size of a single element is returned.

## BIFs and arrays: %ELEM

IBM i

%ELEM(name)

- Returns number of elements in an array or table



```

FIG318.RPGLE X
Line 1      Column 1      Replace
.....DName++++++ETDsFrom+++To/L+++IDc.Keywords+++++
000001    D AnArray      S          5   Dim(10)
000002    D AnotherAry   S          Like(AnArray)
000003    D               Dim(%Elem(AnArray))
000004    D Counter       S          2   0
000005
000006    /Free
000007    ArrayElem = %Elem(AnArray);
000008    // ArrayElem = 10
000009    For Counter = 1 to %Elem(AnotherAry);
000010    // Control in loop based on # elements in AnotherAry (10)
000011    // More logic
000012    EndFor;
000013    *InLR = *on;
000014    /End-free

```

© Copyright IBM Corporation 2009

Figure 3-19. BIFs and arrays: %ELEM

AS075.0

### Notes:

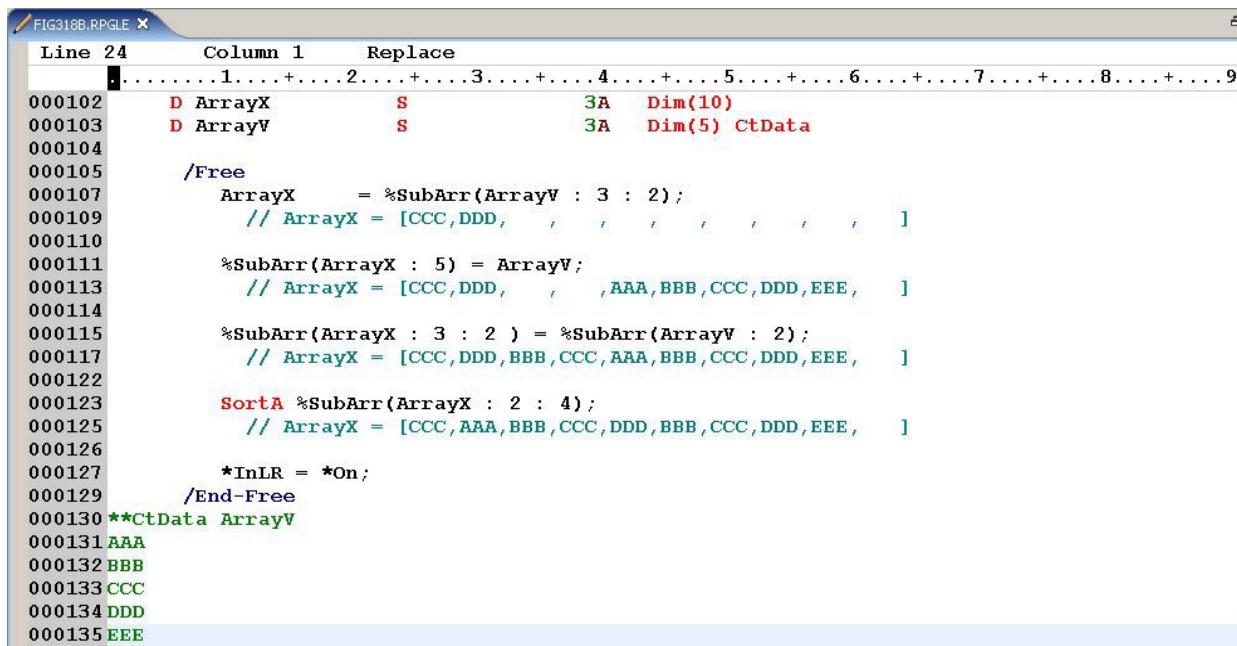
The **%Elem** is similar to the **%Size** function and its benefits in maintenance are the same as those for **%Size**.

# BIFs and arrays: %SUBARR

IBM i

`%SUBARR(name:start-index{:number-of-elements})`

- Get or set a section of the specified array



```

FIG318B RPGLE X
Line 24      Column 1      Replace
000102      D ArrayX          S           3A   Dim(10)
000103      D ArrayV          S           3A   Dim(5) CtData
000104
000105      /Free
000107          ArrayX      = %SubArr(ArrayV : 3 : 2);
000109          // ArrayX = [CCC,DDD,
000110          , , , , , , , , , , ]
000111          %SubArr(ArrayX : 5) = ArrayV;
000113          // ArrayX = [CCC,DDD,
000114          , , , , , ]
000115          %SubArr(ArrayX : 3 : 2) = %SubArr(ArrayV : 2);
000117          // ArrayX = [CCC,DDD,BBB,CCC,AAA,BBB,CCC,DDD,EEE,
000122          , , , , , , , , , ]
000123          SortA %SubArr(ArrayX : 2 : 4);
000125          // ArrayX = [CCC,AAA,BBB,CCC,DDD,BBB,CCC,DDD,EEE,
000126          , , , , , , , , , ]
000127          *InLR = *On;
000129      /End-Free
000130 **CtData ArrayV
000131 AAA
000132 BBB
000133 CCC
000134 DDD
000135 EEE

```

© Copyright IBM Corporation 2009

Figure 3-20. BIFs and arrays: %SUBARR

AS075.0

## Notes:

The built-in function %SUBARR returns a section of the specified array starting at start-index. The number of elements returned is specified by the optional number-of-elements parameter. If not specified, the number-of-elements defaults to the remainder of the array.

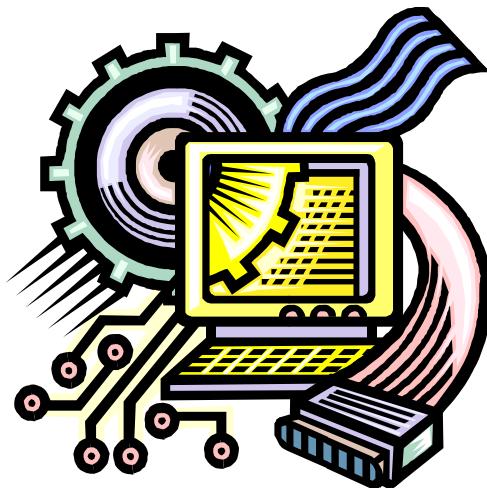
%SUBARR can be used in the following ways:

- On the left-hand side of an assignment using EVAL or EVALR. This changes the specified elements in the specified array.
- Within the expression on the right-hand side of an assignment using EVAL or EVALR where the target of the assignment is an array.
- With the SortA operation.
- As the parameter of the %XFOOT builtin function.

The %SUBARR BIF was an enhancement to the RPG IV language at V5R3.

# Machine exercise: Array processing

IBM i



© Copyright IBM Corporation 2009

Figure 3-21. Machine exercise: Array processing

AS075.0

## Notes:

Perform the machine exercise “Array processing.”

## Unit summary



IBM i

Having completed this unit, you should be able to:

- Define an array in an RPG IV program
- Use the %LookUpxx and %Xfoot BIFs
- Define a table in an RPG IV program

© Copyright IBM Corporation 2009

---

Figure 3-22. Unit summary

AS075.0

### Notes:

# Unit 4. Using data structures and data areas

## What this unit is about

This unit describes how to define and use data structures and data areas in RPG IV.

You learn how to use data structures to redefine data by grouping data fields, and by subdividing data fields. You also see how to build parameter lists for programs using data structures.

You learn how to use data areas to communicate information between programs. You see how to define a data area in an RPG IV program and how to access and update it.

## What you should be able to do

After completing this unit, you should be able to:

- Define a data structure in an RPG IV program
- Name two types of data structures and explain how they would be used
- Define a multidimensional array in an RPG IV program
- Use the Like and LikeDS keywords
- Use Qualified Data Structures
- List one use of a data area
- Define a data area in an RPG IV program

## How you will check your progress

- Machine exercise
  - Students modify an existing program and enhance their code by incorporating a data structure and data area.

## Unit objectives

IBM i

After completing this unit, you should be able to:

- Define a data structure in an RPG IV program
- Name two types of data structures and explain how they would be used
- Define a multidimensional array in an RPG IV program
- Use the Like and LikeDS keywords
- Use qualified data structures
- List one use of a data area
- Define a data area in an RPG IV program

© Copyright IBM Corporation 2009

---

Figure 4-1. Unit objectives

AS075.0

### **Notes:**

# What is a data structure?

IBM i

```
DName++++++ETDsFrom+++To/L+++IDc.Keywords++++Comments....
```

```
D DataStr          DS
D Char13           13A Inz('The Year 2010')
D Char4            4A Overlay(Char13:10)
D Char2            2A Overlay(Char4:3)
D Num4             4S 0 Overlay(Char13:10)
D Exclaim          2A Inz('!!!')
```

T h e        Y e a r        2 0 1 0 ! !

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

<-----DataStr----->

<-----Char13-----| Exclaim>

<----Char4---->

<-Char2->

<----Num4---->

© Copyright IBM Corporation 2009

Figure 4-2. What is a data structure?

AS075.0

## Notes:

A data structure is a way in which you can redefine data. It can be used to group and to subdivide data.

Let's look at a simple example. This visual illustrates one way that you can use data structures and the **overlay** keyword to subdivide data. Suppose we have a field in storage that contains the data *The Year 2010!* We can place this data in a data structure, named **DataStr**, and then subdivide the data structure into its main component fields which are named *Char13* and *Exclaim*. Using the **Overlay** keyword in the D-spec, we define two substrings of the field *Char13* as *Char4* and *Num4*.

The following table shows the values of the data structure, **DataStr**, and its subfields.

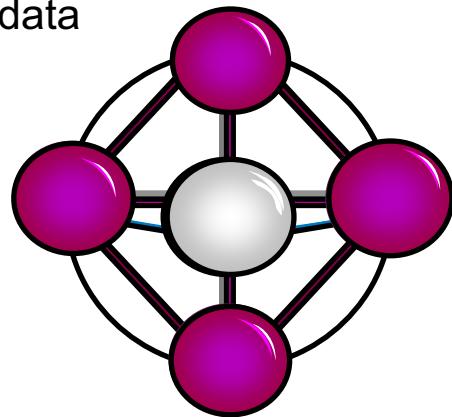
NAME	ATTRIBUTES	VALUE
DATASTR	DS	
CHAR13	CHAR(13)	TheYear2010!!
CHAR2	CHAR(2)	10!!

CHAR4	CHAR(4)	2010''
EXCLAIM	CHAR(2)	!!!
NUM4	ZONED(4,0)	2010.

# Characteristics of data structures

IBM i

- Occupy storage
- Program described or externally described
- Redefines storage
- Internal
- Character string
- Programmer controls arrangement of data



© Copyright IBM Corporation 2009

Figure 4-3. Characteristics of data structures

AS075.0

## Notes:

A data structure is an area in program storage that can be used to define the layout of subfields within that area. It can be program described or externally described.

This visual lists some of the characteristics common to data structures:

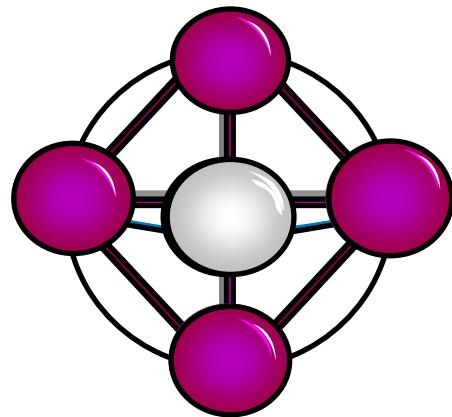
- They occupy storage just like all other data.
- They are created during program initiation and deleted when the program ends.
- The data in a data structure is available only to the program that creates it.
- A data structure permits an RPG IV program to redefine a portion of its variable space. This is not the same as defining fields as we have previously, because each field is assigned a separate storage area, even if more than one field is filled from the same part of a data base record. For example, if the date is retrieved into a field as DATE and also as MO, DAY, and YR, four separate areas are formatted to hold the data. If it is retrieved into a data structure, only one area is assigned and the data is simply reformatted.

- The program treats the data structure as a character string, even if one or more subfields are defined as numeric.

# Types of data structures

IBM i

- Basic
- Multiple occurrence
- Data area
- Program status
- File information



© Copyright IBM Corporation 2009

Figure 4-4. Types of data structures

AS075.0

## Notes:

Basic, Multiple Occurrence, and Data Area types of data structures are covered in this unit.

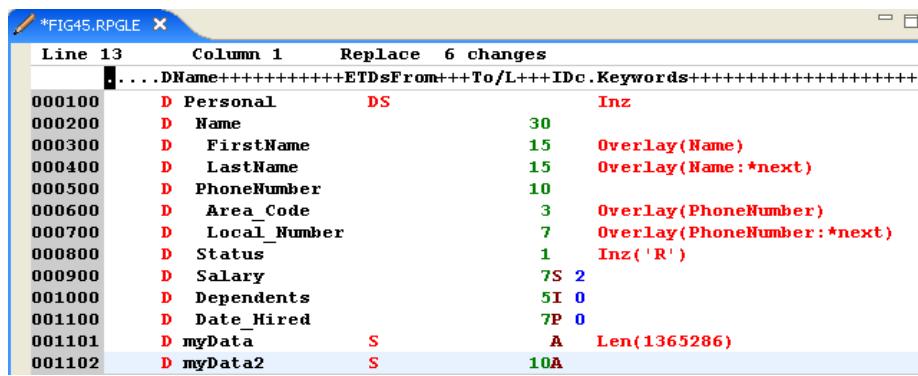
The Program Status and File Information data structures are often used to handle exceptions and are mentioned briefly in the “Managing Exceptions and Handling Errors” unit.

# Defining data structures

IBM i

## Keyword

	<u>Definition</u>
INZ(constant)	Initialize to value of constant
OCCURS(number)	# occurrences in multiple occurrence DS
OVERLAY(name:{pos})	Redefines subfields in a DS
EXTNAME(name{:fmt_name})	File name for externally described DS
EXTFLD(fldname)	External field name being renamed
PREFIX(prefix{nbr_of}char_replaced})	Prefix subfields from externally described DS
LEN(number)	# defining length of field



```
*FIG45.RPGLE X
Line 13      Column 1      Replace 6 changes
. .... DName+++++++
000100  D Personal      DS          Inz
000200  D Name           30
000300  D FirstName      15          Overlay(Name)
000400  D LastName        15          Overlay(Name: *next)
000500  D PhoneNumber     10
000600  D Area_Code       3          Overlay(PhoneNumber)
000700  D Local_Number    7          Overlay(PhoneNumber: *next)
000800  D Status          1          Inz('R')
000900  D Salary          7S 2
001000  D Dependents     5I 0
001100  D Date_Hired     7P 0
001101  D myData          S          A  Len(1365286)
001102  D myData2         S          10A
```

© Copyright IBM Corporation 2009

Figure 4-5. Defining data structures

AS075.0

## **Notes:**

Data structures are defined in your program using the D-spec.

## **Notes:**

- Indented subfield names enables us to show the construct of the data structure clearly and to make the code readable.
- The entire Data Structure is initialized to blanks or zeroes, except **Status**, which is initialized to **R**.
- AreaCode** overlays the first three positions of **PhoneNumber** (positions 31-33 of the Data Structure).
- LocalNumber** overlays the last seven positions of **PhoneNumber** (positions 34-40 of the Data Structure).
- Using **OVERLAY** to *subdivide* or redefine fields is recommended because the maintenance of your code is easier and less error prone.

- The **INZ** keyword can also be used to initialize stand-alone fields and overrides the **INZ** at the Data Structure level.

The rules for using **OVERLAY** are:

- Field overlaid must be a subfield previously defined in this Data Structure.
- Optional **positions** parameter (see the second OVERLAY in the visual) may be a numeric literal, a numeric named constant, or a built-in function.
- Length notation must be used on the overlaying field.
- If an array is overlaid, the **OVERLAY** keyword applies to each element of the array.

Notice the use of **From/To** notation for subfields in the data structure. Also, remember that packed numeric fields defined on the D-Spec have length defined in terms of digits versus the storage required in bytes. The *default numeric type for data structures is zoned decimal*, unless explicitly defined otherwise.

The end of subfield definitions is indicated with the definition of standalone fields or arrays. In this example, **myData** and **myData2** are fields that are not part of the data structure. With V6.1 of IBM i, RPG IV can now have data fields defined as large as 16,773,104 bytes. The **LEN( )** keyword allows you to specify the length of the field. This keyword circumvents the columnar notation 7- digit number limitation. You now have more than enough space to define a field as large as necessary.

Finally, note that a data structure is a character string; so it contains blanks by default. Failure to explicitly initialize all subfields to zero or blank before using them can cause decimal data errors for numeric subfields.

# Program described data structure

Record format:

ItemNo	1	8
CustNo	9	16
Qty	17	20
Price	21	28
Location	29	32
Type	33	36

Subdivide  
the CustNo  
field

D CustNo	DS	2 0
D Region		1
D CustType		
D Number		5 0

What data types are Region and Number??

© Copyright IBM Corporation 2009

Figure 4-6. Program described data structure

AS075.0

## Notes:

This visual shows the description of a program described data structure used to subdivide a field. The same storage area in the program is used to hold **CustNo** and **Region/CustType/Number**. The name **CustNo** refers to the group of subfields described for this data structure.

The entry of **DS** in the Ds columns identifies **CustNo** as a data structure.

The subfields can be alphabetic or numeric. Subfield names cannot be identical to other data structure field names, but they can be the same as fields used elsewhere in the program.

# Redefining fields

**FIG46B1.PF**

Line 1	Column 1	Replace	Functions
000001	A	R ORDERFMTR	TEXT('Order Detail File')
000002	A	ITEMNO	TEXT('Item Number')
000003	A	CUSTNO	TEXT('Customer Number')
000004	A	QTY	4S 0 TEXT('Quantity Ordered')
000005	A	PRICE	8P 2 TEXT('Unit Price')
000006	A	LOCATION	4P 0 TEXT('Location Code')
000007	A	TYPE	4A TEXT('Product Type')

**FIG46B2.RPGLE**

```

Line 17    Column 1    Insert
..... /..1....+....2....+....3....+....4....+....5....+....6....+....7....+....8....+....9.
000100   FOrderFile IF E           Disk
000101
000102   D CustNo      DS
000103   D Region       DS
000104   D CustType     DS
000105   D Number        DS
000108
000109   D OrderDS     DS
000110   D Qty          DS
000111   D Price         DS
000112   D Location      DS
000113
000114   /Free
000115     Read OrderFile;
000116     // (More logic here ....)
000117     *InLR = *On;
000118   /End-Free

```

- What Data Type is PRICE?

© Copyright IBM Corporation 2009

Figure 4-7. Redefining fields

AS075.0

## Notes:

We have coded the example shown in the previous visual. We have explicitly stated the data types for REGION, CUSTTYPE, and NUMBER. Both REGION and NUMBER are defined as zoned decimal. (This is the default numeric data type within a Data Structure.)

We have also coded a second Data Structure ORDERDS to group the QTY, PRICE and LOCATION fields together. Only LOCATION has an explicit data type specified.

What data types are QTY and PRICE defined with in the program?

Both fields are defined as zoned decimal variables in the program. (An excerpt from the compiler listing is shown as follows.)

LOCATION has been explicitly defined as packed decimal in the Data Structure.

If the implicit redefinition of the numeric data type presents a problem (for example, for overlapping subfields), then you should explicitly specify the type. In the next example we see a better way of defining external fields within a Data Structure.

Field	Attributes
*INLR	N(1)

CUSTNO	DS (8)
CUSTTYPE	A(1)
ITEMNO	A(8)
LOCATION	P(4,0)
NUMBER	S(5,0)
ORDERDS	DS(15)
PRICE	S(8,2)
QTY	S(4,0)
REGION	S(2,0)
TYPE	A(4)

# Externally described data structure

IBM i

- For externally described files only
- No F-spec declaration required

```

Session B - [24 x 80]
File Edit View Communication Actions Window Help
Print Copy Paste Send Recv Display Color Map Record Stop Play Quit Clipbrd Support Index

Display Spooled File
File . . . . . : FIG47A          Page/Line   2/6
Control . . . . . : + 2           Columns     1 - 78
Find . . . . .
*....+....1....+....2....+....3....+....4....+....5....+....6....+....7....+...
Source Listing
1 D ItemDS      E DS             ExtName(Item_PF:Item_Fmt)
2 /Free
*-----
* Data structure . . . . . : ITEMDS
* External format . . . . . : ITEM_FMT : AS07V2LIB/ITEM_PF
* Format text . . . . . : Item Master Record
*-----
3=D ITMNBR          5P 0
4=D ITMDESCR        25A
5=D ITMQTYOH        7P 0
6=D ITMQTYOO        7P 0
7=D ITMCOST          5P 2
8=D ITMPRICE         5P 2
9=D VNDNBR          5P 0
10=D ITMVNDCAT#       7A
More...
F3=Exit   F12=Cancel  F19=Left   F20=Right   F24=More keys

```

Figure 4-8. Externally described data structure

AS075.0

## Notes:

Some advantages of an externally described data structure include:

- Maintains integrity of your data definitions
- Reduces the coding required

In a program, there is no functional difference between a program described and an externally described data structure. An externally described data structure might also be the description of a database file, so it can help to maintain the integrity of data definitions.

If a data structure is to be used in multiple programs, externally describing it can reduce the coding required. The data structure can be described once, and then referenced whenever it is needed. This can be particularly useful when a data structure is used to pass a parameter list between programs.

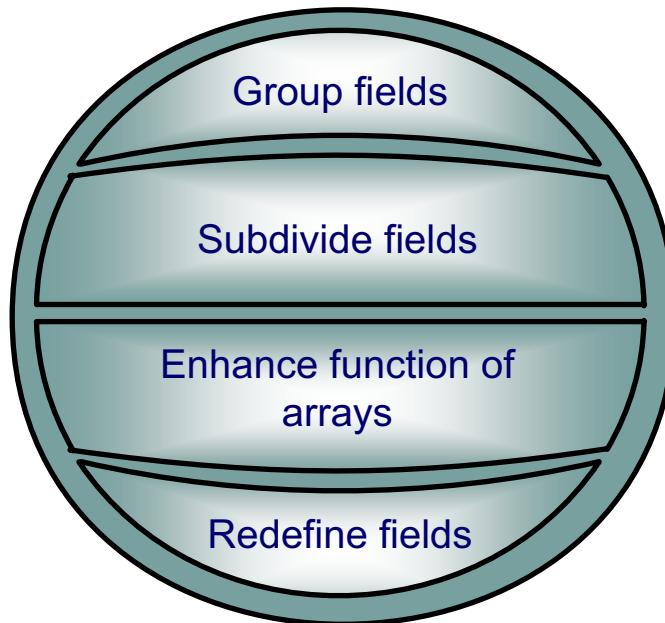
Three steps are required to externally describe a data structure:

1. The first step is to describe DDS for a physical file. This DDS might already exist.

2. The second step is to create the physical file. Again it might already exist, and the file might or might not contain data. This file description becomes the external description of the data structure. When the physical file is created, if the **member parameter** is **\*NONE**, there are no data members associated with it, and data is part of the file only while it is being used as a data structure.
3. In step number three, the program describes the data structure by referencing the physical file name on the Data Definition Specifications. The data structure name is entered starting in column 7 or after. **DS** in the **Ds** columns identifies this as a data structure. The **E** in column 22 indicates that it is externally described, and the name of the physical file that supplies the description is supplied as a parameter of the **EXTNAME** keyword (the format name is optional).

# Uses for data structures

IBM i



© Copyright IBM Corporation 2009

Figure 4-9. Uses for data structures

AS075.0

## Notes:

The visual summarizes some uses of data structures:

- **Group fields:** The fields might be from one or more input records and are not contiguous.
- **Subdivide fields:** You can break up a field into component parts, for example, a field called **FullName** into **FirstName** and **LastName**.
- **Arrays:** When defined as part of a data structure, we can create arrays of more than a single dimension (a matrix).
- **Redefine fields:** You can use a data structure to redefine fields with other names (and restructure).

# Grouping fields

IBM i

Given the following record:

OrderNo	1	5
PONumber	6	10
OrderDate	11	16 0
SalesPerson	17	20 0
OperInit	21	23

We want to group several noncontiguous fields:

The screenshot shows the IBM i CODE editor interface. The title bar reads "CODE - <OS400A>AS07U2LIB/QRPGLESRC(FIG49)". The menu bar includes File, Edit, View, Actions, Options, Windows, Help, and Extras. A toolbar with various icons is above the main window. The main window displays a resequencing command:

```

Row 5      Column 1      Replace
.....DName++++++ETDsFrom+++To/L+++IDc.Keywords+++++*
000001    D Stamp      DS          Inz
000002    D OrderNo    5
000003    D OrderDate  6  0
000004    D OperInit   3
000005
AS07U2LIB/QRPGLESRC(FIG49) resequenced and saved

```

Below the command, a status message says "AS07U2LIB/QRPGLESRC(FIG49) resequenced and saved". At the bottom of the window, there is a copyright notice: "© Copyright IBM Corporation 2009".

Figure 4-10. Grouping fields

AS075.0

## Notes:

In this example, the data structure **Stamp** is used to group together three fields that are separated by other fields in a record. The space occupied by these fields is redefined so that when the order number, **OrderNo**, changes, the data structure, **Stamp**, changes as well.

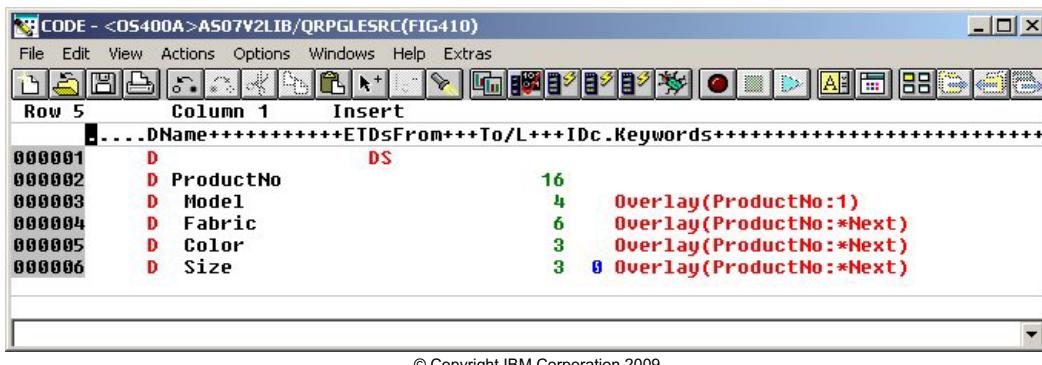
# Subdividing fields

IBM i

Given the record format:

ProductNo	1	16
ProdName	17	40
ProdSeason	41	46
ProdPrice	47	52 2

Subdivide the ProductNo field:



The screenshot shows the IBM i CODE editor interface with the title bar "CODE - <OS400A>A507V2LIB/QRPGLESRC(FIG410)". The menu bar includes File, Edit, View, Actions, Options, Windows, Help, and Extras. The toolbar has various icons for file operations like Open, Save, Print, and Insert. A status bar at the bottom right says "AS075.0". The code area displays the following definition:

```

D DName+++++ETDsFrom++To/L+++IDc.Keywords+++++
  . . .
  000001  D DS
  000002  D ProductNo      16
  000003  D Model          4   Overlay(ProductNo:1)
  000004  D Fabric          6   Overlay(ProductNo:*Next)
  000005  D Color           3   Overlay(ProductNo:*Next)
  000006  D Size            3   8 Overlay(ProductNo:*Next)
  . . .

```

Copyright IBM Corporation 2009

Figure 4-11. Subdividing fields

AS075.0

## Notes:

In this example, **ProductNo** is made up of *Model*, *Fabric*, *Color*, and *Size*. The field, **ProductNo**, is updated when a record is read from the input file, and then broken into its subfields using a data structure. Length notation is used in this example. That is, we specify field lengths only.

Alternatively, we could specify from/to positions (also known as from/to notation). The subfields can be used anywhere in the program.

Specifying **OVERLAY(name:\*****NEXT**) in the keywords positions the subfield at the next available position within the overlaid field. This is the first position after all other subfields, prior to this subfield, that overlay the same subfield.

# Using a data structure and named indicators

IBM i

```

....+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
000001 // Declare Files
000002 FItem_PF IF E          K Disk
000003 >1 FItemInq2 CF E      Workstn IndDS(WkstnInd)
000004
000005 // Map indicators in DSPF to named indicators
000006 >2 D WkstnInd DS
000007 D NotFound           40   40N
000008 D LowQty             30   30N
000009 >3 D Exit              03   03N
000010
000011 /FREE
000012 PgmNam = 'ITEMINQOU';
000013 Write Header; // Write to buffer
000014 Write Footer; // Write to buffer
000015 ExFmt Prompt; // Write to buffer; write buffer to display; enable read
000016
000017 >4 Dow NOT Exit;
000018   Chain ItmNbr Item_PF;
000019   NotFound = Not %Found(Item_PF); // Set indicator for record not found
000020
000021   If %Found(Item_PF); // Item Number valid?
000022     LowQty = (ItmQty0H + ItmQty00) < 20; // Set Indicator for Qty < Minimum
000023     Write Detail; // Write to buffer
000024   Endif;
000025
000026   // Display prompt with error or not
000027   ExFmt Prompt; // Write to buffer; write buffer to display; enable read
000028 Enddo;
000029 // F3 pressed; user wants to exit program
000030 *InLR = *on;
000031 /END-FREE

```

© Copyright IBM Corporation 2009

Figure 4-12. Using a data structure and named indicators

AS075.0

## Notes:

We have been using the ability to define an indicator as a named variable. This capability allows the use of a variable name rather than an indicator in an RPG IV program. There are some things that you must consider:

- Named indicators are supported in RPG IV programs only. They are **not** supported in DDS PRTFs nor DSPFs.
- A popular technique is to use a data structure to map specific numeric indicators to named indicators.
- If you map an indicator that is used by a DSPF, for example, you would use the INDARA keyword to correlate the DSPF (or PRTF) indicators with the indicator data structure.

There are several points to notice in the visual:

- To establish the mapping of the numeric indicators in the display file, **ItemInq2**, you must code the **INDARA** keyword at the file level in the DDS. And, you must code the **IndDS** keyword, naming a data structure in the F-spec keyword (in this case, **WkstnInd**) that map the named indicators to the numeric ones.

2. The name of the data structure *must match* the one you coded in the F-spec.
3. *Each and every indicator* in the display file is mapped to a named indicator. When the numeric indicator is mapped, you cannot use the numeric indicator in your program. This should not be a problem for us, because all we use are named indicators.
4. Named indicators are more easily understood and can be operated on and set just like the legacy named indicators that we replace.

There is another solution that is referenced in the *Who Knew You Could Do That with RPG IV?* Redbook that involves overlaying only those specific indicators that your program will reference. To do this, you must use what is known as a based variable. We will not be discussing based variables and the associated pointer data type further. We discuss them in the advanced class that follows this one (AS10/S6199). If you want to investigate it on your own, the Redbook is an excellent reference.

# Qualified names

IBM i

**DsName.SubFldName**

- Simple qualified name

**Ds1Name.Ds2Name.Ds3Name.SubFldName**

- Compound qualified name

- QUALIFIED keyword on DS
- LIKEDS keyword to define one DS like another DS
- LIKEDS with DS also says QUALIFIED
- Subfields in several DSs can have the same names

© Copyright IBM Corporation 2009

Figure 4-13. Qualified names

AS075.0

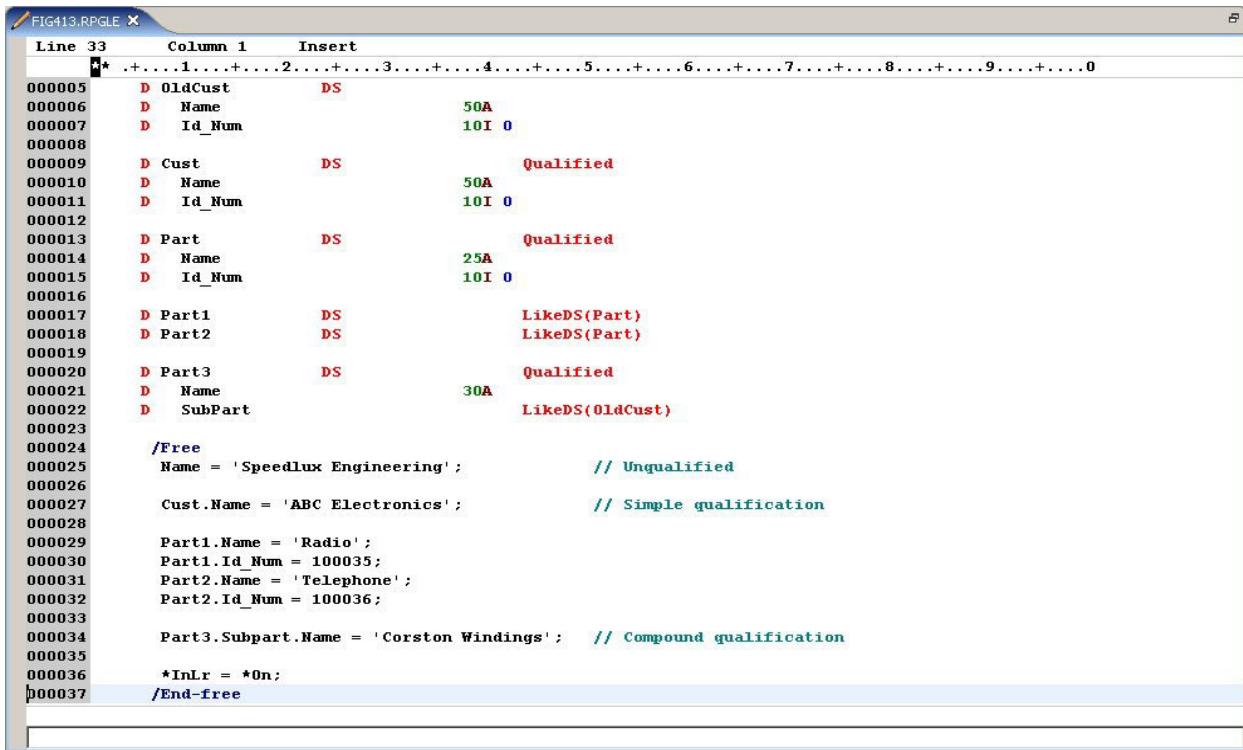
**Notes:**

This visual overviews the features of qualified naming.

Qualified naming allows us much more flexibility. Subfields in two different data structures can have the same names with completely different attributes.

# QUALIFIED and LIKEDS: Example

IBM i



```

FIG413.RPGLE X
Line 33    Column 1    Insert
* .+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8....+....9....+....0
000005    D OldCust      DS
000006    D  Name          50A
000007    D  Id_Num        10I 0
000008
000009    D Cust          DS      Qualified
000010    D  Name          50A
000011    D  Id_Num        10I 0
000012
000013    D Part          DS      Qualified
000014    D  Name          25A
000015    D  Id_Num        10I 0
000016
000017    D Part1         DS      LikeDS(Part)
000018    D Part2         DS      LikeDS(Part)
000019
000020    D Part3         DS      Qualified
000021    D  Name          30A
000022    D  SubPart        LikeDS(OldCust)
000023
000024 /Free
000025   Name = 'Speedlux Engineering';           // Unqualified
000026
000027   Cust.Name = 'ABC Electronics';           // Simple qualification
000028
000029   Part1.Name = 'Radio';
000030   Part1.Id_Num = 100035;
000031   Part2.Name = 'Telephone';
000032   Part2.Id_Num = 100036;
000033
000034   Part3.Subpart.Name = 'Corston Windings'; // Compound qualification
000035
000036   *InLr = *On;
000037 /End-free

```

© Copyright IBM Corporation 2009

Figure 4-14. QUALIFIED and LIKEDS: Example

AS075.0

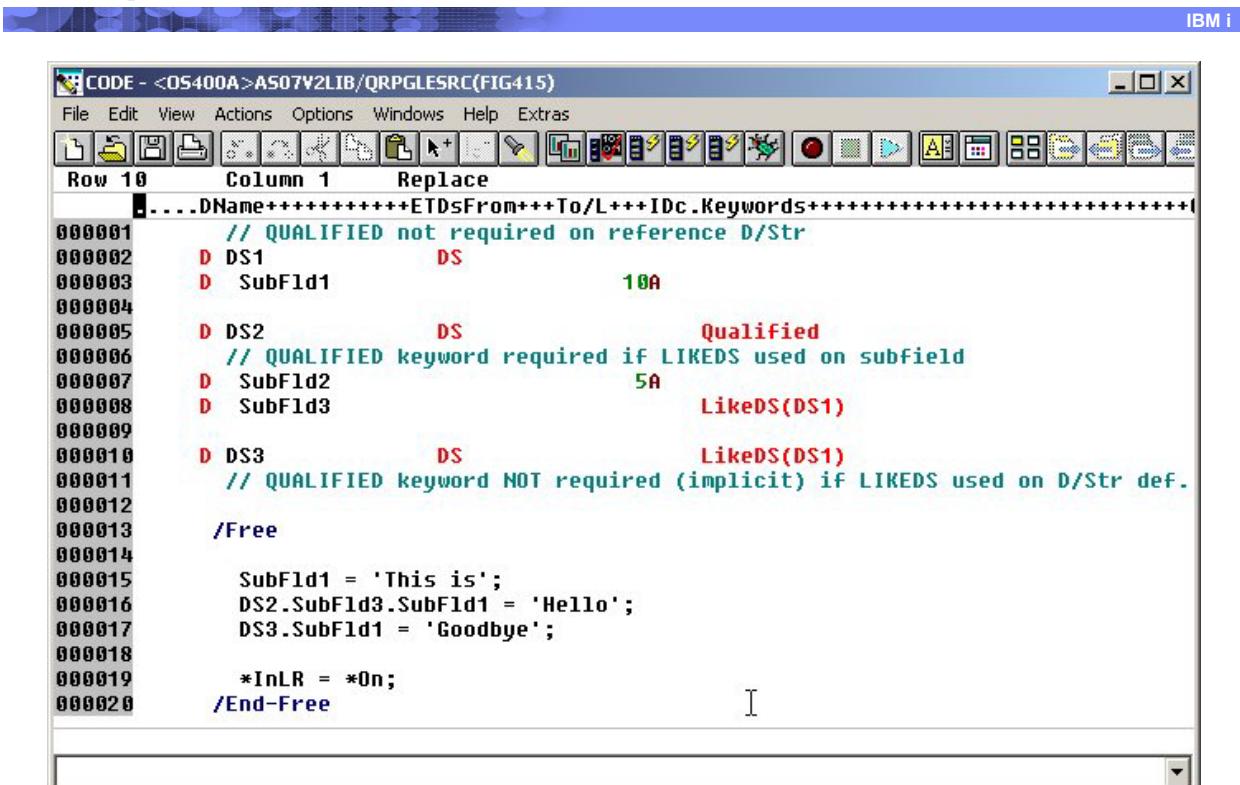
## Notes:

This example illustrates the use of the **LIKEDS** and **QUALIFIED** keywords for data structures.

- The **Cust** and **Part** data structures are **QUALIFIED**. Thus references to subfields must be use qualified naming format as shown. Notice that these two DSs have a subfield with the same name.
- The **Part1** and **Part2** data structures are *secondary* data structures. They need only the **LIKEDS** keyword and no **QUALIFIED** keyword. Because its subfields will be named the same as those of the data structure from which they are derived, qualified naming is mandatory and the **QUALIFIED** keyword is not required.
- In calculations, you must reference the data structure fields using qualified names or you get an error.

Note that **OldCust** is an unqualified data structure. It can be referred to for qualifying **Part3**. The **Part3** data structure is an example of compound qualification.

# Compound data structures



The screenshot shows a code editor window titled "CODE - <OS400A>AS07V2LIB/QRPGLESRC(FIG415)". The window has a toolbar with various icons at the top. Below the toolbar, the code is displayed in a text area:

```

File Edit View Actions Options Windows Help Extras
Row 10 Column 1 Replace
.....DName++++++ETDsFrom++To/L+++IDc.Keywords+++++
000001 // QUALIFIED not required on reference D/Str
000002 D DS1          DS
000003 D SubFld1      10A
000004
000005 D DS2          DS          Qualified
000006 // QUALIFIED keyword required if LIKEDS used on subfield
000007 D SubFld2      5A
000008 D SubFld3      LikeDS(DS1)
000009
000010 D DS3          DS          LikeDS(DS1)
000011 // QUALIFIED keyword NOT required (implicit) if LIKEDS used on D/Str def.
000012
000013 /Free
000014
000015     SubFld1 = 'This is';
000016     DS2.SubFld3.SubFld1 = 'Hello';
000017     DS3.SubFld1 = 'Goodbye';
000018
000019     *InLR = *On;
000020 /End-Free

```

© Copyright IBM Corporation 2009

Figure 4-15. Compound data structures

AS075.0

## Notes:

A data structure can contain another data structure as shown in this example. Notice the qualified naming in the example. Look particularly at:

`DS2.SubFld3.SubFld1 = Hello';'`

Because **SubFld1** is a subfield of **SubFld2**, a data structure that is defined like **DS1**, we reference the subfield of the **DS2** data structure in this fashion.

# LIKEREc keyword

The screenshot shows two code snippets in the IBM i editor.

**Top Snippet (POPNLI\_LF.LF):**

```

Line 1      Column 1      Replace
. ....+A*..1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
000100    A*****+
000200    A*  PO Open Line Item LF:  POOPNLI_LF
000300    A*****+
000400    A
000500    A          R  POLINE_FMT
000600    A
000700    A          K  PONBR
000800    A          K  ITMNBR

```

**Bottom Snippet (FIG414B.RPGLE):**

```

Line 1      Column 1      Replace
. ....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8....+....9.
000100
000101    FPOOpnLI_LF IF   E           K Disk
000200
000300    D KeyStr        DS          LikeRec(POLine_Fmt:*Key)
000400
000500    /Free
000600    KeyStr.PONbr = 300001;
000601    KeyStr.ItmNbr = 20002;
000602
000700    Chain %Kds(KeyStr) POOpnLI_LF;
000800
000900    If %Found(POOpnLI_LF);
001000    Dsply PolitmCost '*REQUESTER';
001100    Endif;
001200
001300    *InLR = *On;
001400    /End-free

```

Figure 4-16. LIKEREc keyword

AS075.0

## Notes:

This feature enables you to describe a DS based upon a record format. Any record format used in this way must be declared on an F-spec in your program.

Notice several things in this example:

- We are using a DS to extract a key for the **POLine\_Fmt** record format. Note the **\*Key** parameter. This parameter extracts only those fields that make up the record key in the order that they are defined in the K specs of the file's DDS. There are several other options:
  - **\*ALL**: All fields in the external record are extracted.
  - **\*INPUT**: All input-capable fields are extracted. This is the default.
  - **\*OUTPUT**: All output-capable fields are extracted.
- When processing the record using the %KDS BIF, you can reference the DS where the key is defined. %KDS is allowed as the search argument for any keyed Input/Output operation (CHAIN, DELETE, READE, READPE, SETGT, SETLL) coded in a free-form group.

# Arrays within data structure

IBM i

ItemNo	Desc	QtyOH	QtyOO	Sold
20001	Telephone, one line	10	6	100
20002	Telephone, two line	5	12	560
20003	Speaker Telephone	6	8	25
20004	Telephone Extension Cord	25	10	56
20005	Dry Erase Marker Packs	428	10	1200
20006	Executive Chairs	10	2	15
20007	Secretarial Chairs	13	5	156
20008	Desk Calendar Pads	56	10	1543
20009	Diskette Mailers	128	200	234

© Copyright IBM Corporation 2009

Figure 4-17. Arrays within data structure

AS075.0

## Notes:

We discussed arrays earlier. An *array* is defined as a group of fields of the *same data type*. A *data structure* is a grouping of fields that can be a combination of *one or more data types*. A data structure could be compared to a record. There are two ways to define arrays within data structures.

1. A *Multiple Occurrence Data Structure* is a *grouping of data structures* that are defined the same, like, a grouping of records. You could compare a multiple occurrence data structure to a subfile that is used internally in the program. We discuss subfiles in the next unit.
2. An *Array Data Structure* (available since in V5R2 of ILE RPG) is a data structure defined with keyword DIM. An array data structure is like a multiple-occurrence data structure, except that the index is explicitly specified, as with arrays.

This visual shows a group of records that we will describe as an array within a data structure.

# Using arrays within data structures

IBM i

- Multiple copies of a single data structure
- Old method (MOD/S):
  - Specify OCCURS on D-Spec
  - Use %OCCUR BIF in calculations
- New method (Array):
  - Specify DIM on D-Spec for array data structure
  - Use array indexing in calculations

© Copyright IBM Corporation 2009

Figure 4-18. Using arrays within data structures

AS075.0

## Notes:

A multiple occurrence data structure is based upon a single data structure definition with multiple occurrences. Each occurrence has the same format and attributes, although unlike arrays, the subfields do not have to have the same lengths and attributes. Each copy of the multiple occurrence data structure is accessed individually, by referring to its occurrence number.

Identifying a multiple occurrence data structure requires a special entry **OCCURS** on the data definition spec. When specified, the occurrence of the data structure to be used is identified with the **%OCCUR** built-in function opcode, and processing is done as if there were only one occurrence.

A multiple occurrence data structure can also simplify coding for a *multidimensional* array if one is required. However, since V5R2, defining and using array data structures have been much simpler.

## Pre-V5R2: Multiple occurrence data structure

The screenshot shows two windows from the IBM i code editor.

**Top Window (D-Spec):**

```
CODE - <AECAUX>AS07Y2LIB/QRPGLESRC(FIG418A)
File Edit View Actions Options Windows Help
Row 3 Column 10 Replace
-----DName-----+-----ETDsFrom---To/L+++IDc.Keywords-----
000001  Inventory   DS
000002  D ItemNo
000003  D Desc
000004  D Oty0H
000005  D Oty0O
000006  D Sales
000007  =           6S 0
                    25
                    5S 0
                    5S 0
                    5S 0
                    5S 0
```

**Bottom Window (RPG Code):**

```
CODE - <AECAUX>AS07Y2LIB/QRPGLESRC(FIG418B)
File Edit View Actions Options Windows Help
<AECAUX>AS07Y2LIB/QRPGLESRC(FIG418B)
Row 2 Column 1 Insert
-----1-----2-----3-----4-----5-----6-----7-----
000001 /Free
000002 %Occur(Inventory) = 3;
000003 TotSales += Sales;
000004 /End-free
000005
```

A callout box highlights the multiple occurrences of the **Inventory** structure in the D-Spec, with an arrow pointing to the corresponding loop control in the RPG code.

Figure 4-19. Pre-V5R2: Multiple occurrence data structure

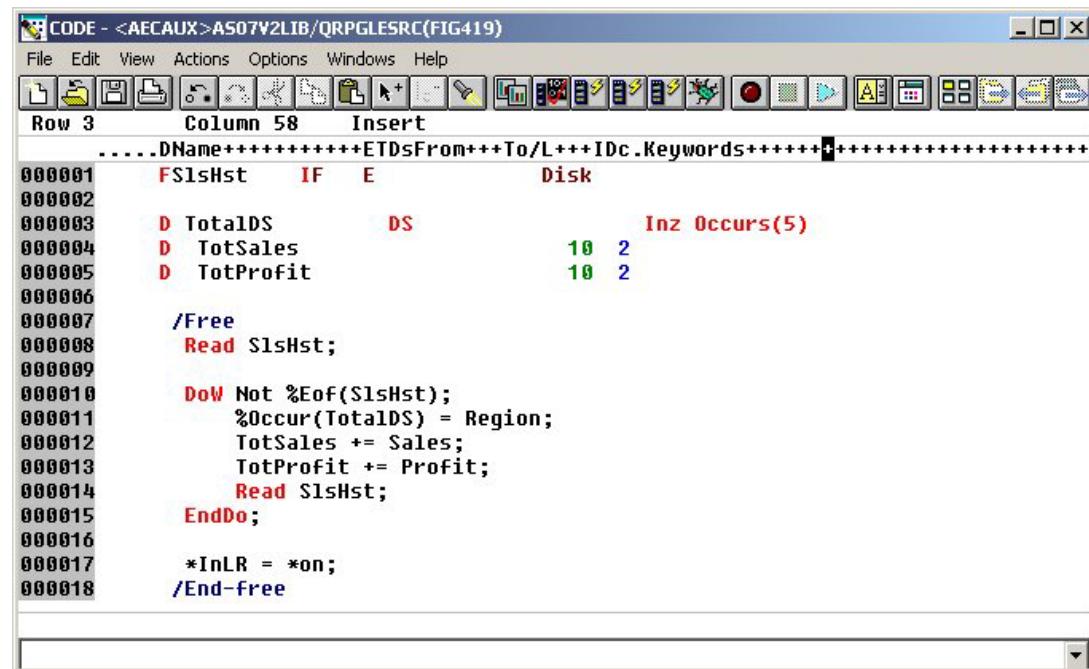
AS075.0

### Notes:

This visual is the coding for the group of records in a prior visual. The **OCCURS** keyword in the D-Spec specifies how many groups of records are defined with in the data structure. The data structure is defined as before and, although we have shown length notation, you can use absolute from/to notation.

We use the **%Occur** BIF to manage which occurrence we process.

# Processing occurrences of MOD/S



```

CODE - <AECAUX>AS07V2LIB/QRPGLESRC(FIG419)
File Edit View Actions Options Windows Help
Row 3 Column 58 Insert
.....DName++++++ETDsFrom+++To/L+++IDc.Keywords+++++*+++++*+++++*+++++
000001   FSlsHst    IF   E           Disk
000002
000003   D TotalDS      DS          Inz Occurs(5)
000004   D TotSales      10  2
000005   D TotProfit     10  2
000006
000007   /Free
000008     Read SlsHst;
000009
000010   DoW Not %Eof(SlsHst);
000011     %Occur(TotalDS) = Region;
000012     TotSales += Sales;
000013     TotProfit += Profit;
000014     Read SlsHst;
000015   EndDo;
000016
000017   *InLR = *on;
000018 /End-Free

```

© Copyright IBM Corporation 2009

Figure 4-20. Processing occurrences of MOD/S

AS075.0

## Notes:

In this example, the **SLSHST** file is used to record sales history by region. The **SLSHST** record includes region, sales, and profit fields in addition to other data. **TotalDS** is a multiple occurrence data structure that captures the sales and profit information for each of five regions.

To identify **TotalDS** as a multiple occurrence data structure, we coded the **OCCURS** keyword on the data definition specifications line that defines the data structure. The value **5** represents the number of times that the data structure is to be repeated.

The BIF, **%Occur**, is used to manage which occurrence is processed. The occurrence is determined by the value of the field **Region** stored in each record. **%Occur** can either be set in an expression (as shown) or can be on the right side and set a value of a field.

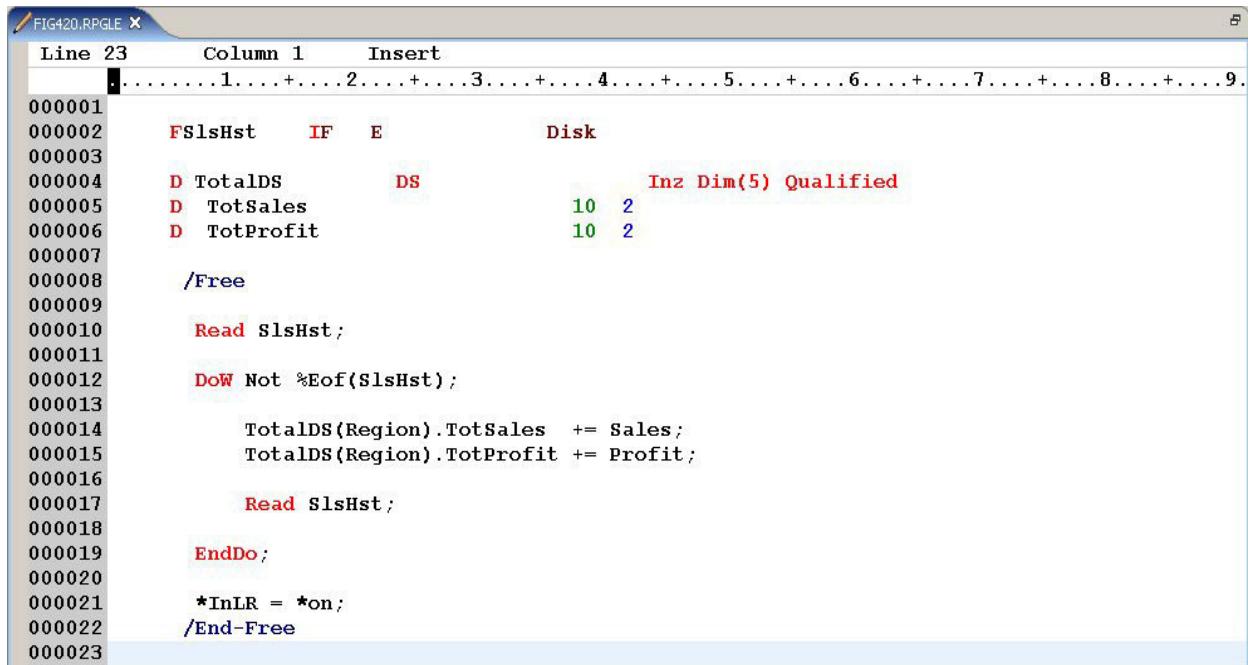
All other processing is done as if there were only one set of fields being maintained.

We have five regions in our operation and that we are maintaining sales and profit totals for each region one year at a time.

A multiple occurrence data structure is actually multiple instances of a data structure. Each instance has the same format and attributes, although unlike arrays the subfields do not have to have the same lengths and attributes. Each instance of the multiple occurrence data structure is accessed individually, by referring to its occurrence number.

## V5R2: Array data structure

IBM i



```

FIG420.RPGLE X
Line 23      Column 1      Insert
. .... 1....+....2....+....3....+....4....+....5....+....6....+....7....+....8....+....9.
000001
000002     FSlsHst    IF   E           Disk
000003
000004     D TotalDS       DS          Inz Dim(5) Qualified
000005     D TotSales
000006     D TotProfit
000007
000008     /Free
000009
000010     Read SlsHst;
000011
000012     DoW Not %Eof(SlsHst);
000013
000014     TotalDS(Region).TotSales  += Sales;
000015     TotalDS(Region).TotProfit += Profit;
000016
000017     Read SlsHst;
000018
000019     EndDo;
000020
000021     *InLR = *on;
000022     /End-Free
000023

```

© Copyright IBM Corporation 2009

Figure 4-21. V5R2: Array data structure

AS075.0

### Notes:

This example is a newer way of coding the example from the previous visual. Notice that the DIM keyword applies to the whole data structure. Also, notice the **QUALIFIED** keyword. Again, we read each record of the SlsHst file, which gives us the **Sales** and **Profit** fields in addition to the **Region** that made the sale and profit.

By using the Region as the index to the arrays, and using qualified naming, we calculate the total sales and profit for each region read from the **SlsHst** file.

Notice also the use of the abbreviated assignment operator **+=**. In this situation, this form saves you having to code the qualified name of the array element twice.

# Array as a component of a data structure

```

FIG420B.RPGLE X
Line 29      Column 1      Insert
1.....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8....+....9.
000001
000002      FSlsHst      IF     E           Disk
000003
000004      D  TotalDS       DS           Inz Qualified
000005      D  Totals        DS           20A   Dim(5)
000006      D  TotSales      DS           10S  2 Overlay(Totals)
000007      D  TotProfit     DS           10S  2 Overlay(Totals:11)
000008
000009      /Free
000010
000011      Read  SlsHst;
000012
000013      DoW  Not %Eof(SlsHst);
000014
000015          TotalDS.TotSales(Region)  += Sales;
000016          TotalDS.TotProfit(Region) += Profit;
000017
000018          Read  SlsHst;
000019
000020      EndDo;
000021
000022      // Sort by Profit sub-array - Sales association retained
000023      SortA TotalDS.TotProfit;
000024
000025      *InLR = *on;
000026
000027      /End-Free
000028
000029
000030
000031

```

© Copyright IBM Corporation 2009

Figure 4-22. Array as a component of a data structure

AS075.0

## Notes:

Suppose we had an application where we had an array that we wanted to divide into subfields. How could we do this? We know that the OVERLAY keyword can be used in a data structure to subdivide a field into subfields. We can also make an array part of a data structure and thereby subdivide elements of the array into subfields using OVERLAY.

The visual is a variation of the previous example. The DIM keyword has been placed against the **Totals** subfield within the data structure. Two further subfields overlay **Totals**, allowing the SORTA operation to be used with either **TotSales** or **TotProfit**. The association between **TotSales** and **TotProfit** is maintained.

# Multidimensional array

IBM i

Array (SALES) has 12 elements.

Each element contains sales for a calendar month.

Sales(1)	January Sales
Sales(2)	February Sales
Sales(3)	March Sales
Sales(4)	April Sales
:	
:	
Sales(12)	December Sales

## OLD METHOD

```
D SalesByRegion    DS          Inz Occurs(5)
D   Sales           7  2 Dim(12)
```

## NEW METHOD

```
D SalesByRegion    DS          Dim(5) Qualified
D   Sales           7  2 Dim(12)
```

© Copyright IBM Corporation 2009

Figure 4-23. Multidimensional array

AS075.0

## Notes:

In a previous unit, we discussed how to build a single dimensional array. This example expands that capability, using a multiple occurrence data structure (V5R1) to create a multidimensional array.

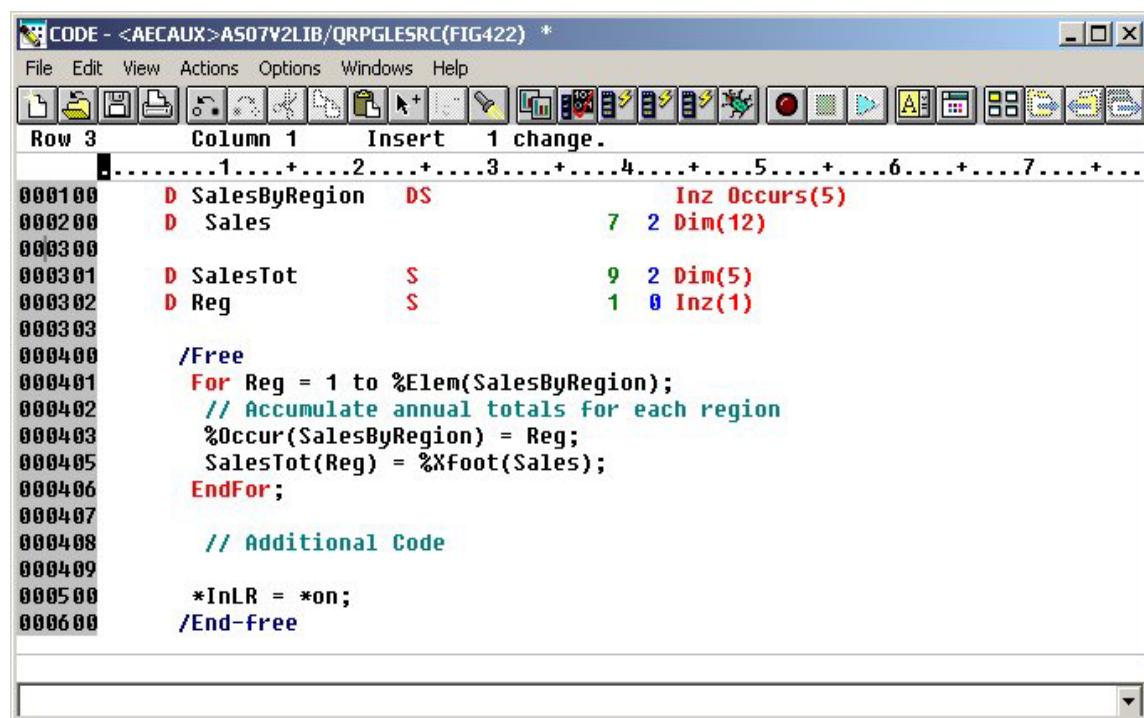
In this example, we show the use of an array in a Multiple Occurrence Data Structure or **MODS** Elements can be summed across an array, or summed across a D/S. We have built a 2-dimensional array.

In this case, the array Sales has 12 elements, each containing Sales values for each month. The **MODS** allows us to track Sales by Region. Thus, Sales can be summed for the year by region.

Notice the **OCCURS(5)** in the keywords area of the D-spec defining the data structure and one set of the Sales array for each region.

Since V5R2, the OCCURS keyword is not used. Instead, a qualified data structure is used with the DIM keyword. This method is much simpler than the older method.

## Pre-V5R2: Multidimensional array



The screenshot shows the IBM i code editor interface. The title bar reads "CODE - <AECAUX>AS07V2LIB/QRPGLESRC(FIG422) \*". The menu bar includes File, Edit, View, Actions, Options, Windows, and Help. A toolbar with various icons is above the code area. The code itself is written in RPG IV and defines a multidimensional array:

```

Row 3      Column 1      Insert  1 change.
.....1.....+.....2.....+.....3.....+.....4.....+.....5.....+.....6.....+.....7.....+...
000100    D SalesByRegion  DS          Inz Occurs(5)
000200    D Sales           7 2 Dim(12)
000300
000301    D SalesTot        S          9 2 Dim(5)
000302    D Reg             S          1 0 Inz(1)
000303
000400    /Free
000401    For Reg = 1 to %Elem(SalesByRegion);
000402        // Accumulate annual totals for each region
000403        %Occur(SalesByRegion) = Reg;
000404        SalesTot(Reg) = %Xfoot(Sales);
000405    EndFor;
000406
000407
000408    // Additional Code
000409
000500    *InLR = *on;
000600    /End-Free

```

© Copyright IBM Corporation 2009

Figure 4-24. Pre-V5R2: Multidimensional array

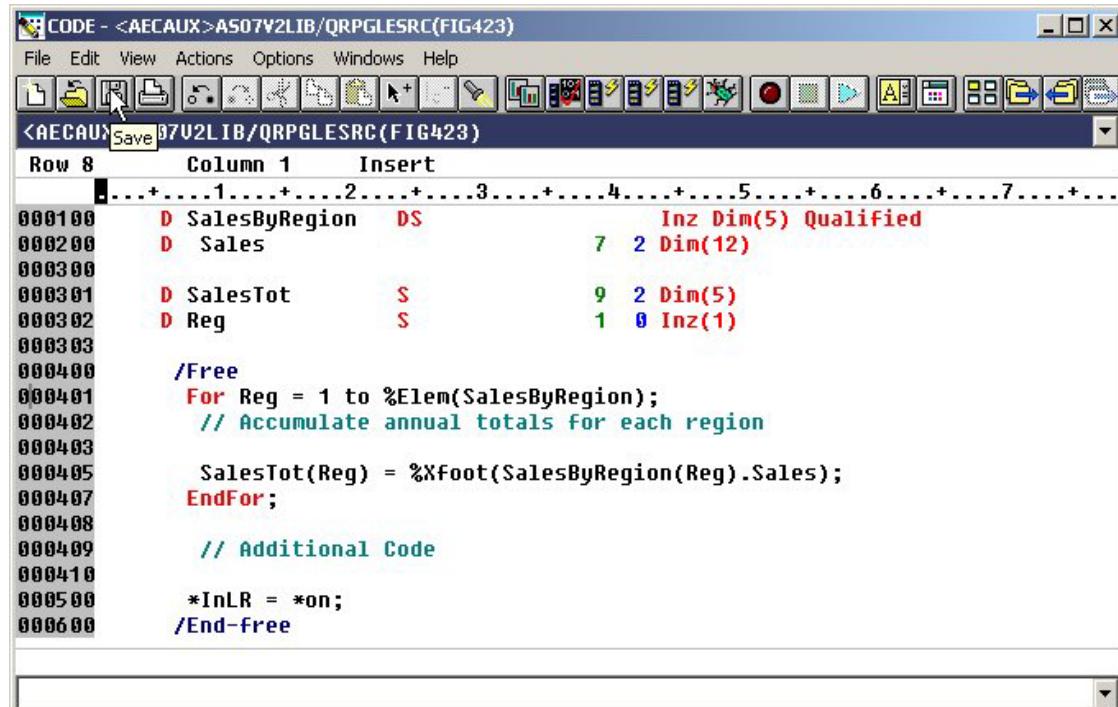
AS075.0

### Notes:

Using the **%OCCUR** BIF, the program selects the appropriate data structure occurrence. Because there are five regions, the value of **Reg** ranges from 1 to 5. Each time the value of **Reg** is incremented, the program selects the appropriate occurrence in the data structure and accumulates annual sales for that region.

For multiple occurrence data structures, all occurrences are initialized based upon the data type. You should always use the **INZ** keyword to initialize a data structure and override at the subfield level as required.

## V5R2: Multidimensional array



The screenshot shows the IBM i code editor interface. The title bar reads "CODE - <AECAUX>AS07V2LIB/QRPGLESRC(FIG423)". The menu bar includes File, Edit, View, Actions, Options, Windows, and Help. A toolbar with various icons is above the main window. The code editor window has "Row 8" and "Column 1" labels at the top left. The code itself is as follows:

```


Row 8    Column 1      Insert
000100  D SalesByRegion  DS          Inz Dim(5) Qualified
000200  D Sales           DS          7 2 Dim(12)
000300
000301  D SalesTot        S          9 2 Dim(5)
000302  D Reg             S          1 0 Inz(1)
000303
000400  /Free
000401    For Reg = 1 to %Elem(SalesByRegion);
000402      // Accumulate annual totals for each region
000403
000405      SalesTot(Reg) = %Xfoot(SalesByRegion(Reg).Sales);
000407    EndFor;
000408
000409    // Additional Code
000410
000500    *InLR = *on;
000600  /End-free


```

© Copyright IBM Corporation 2009

Figure 4-25. V5R2: Multidimensional array

AS075.0

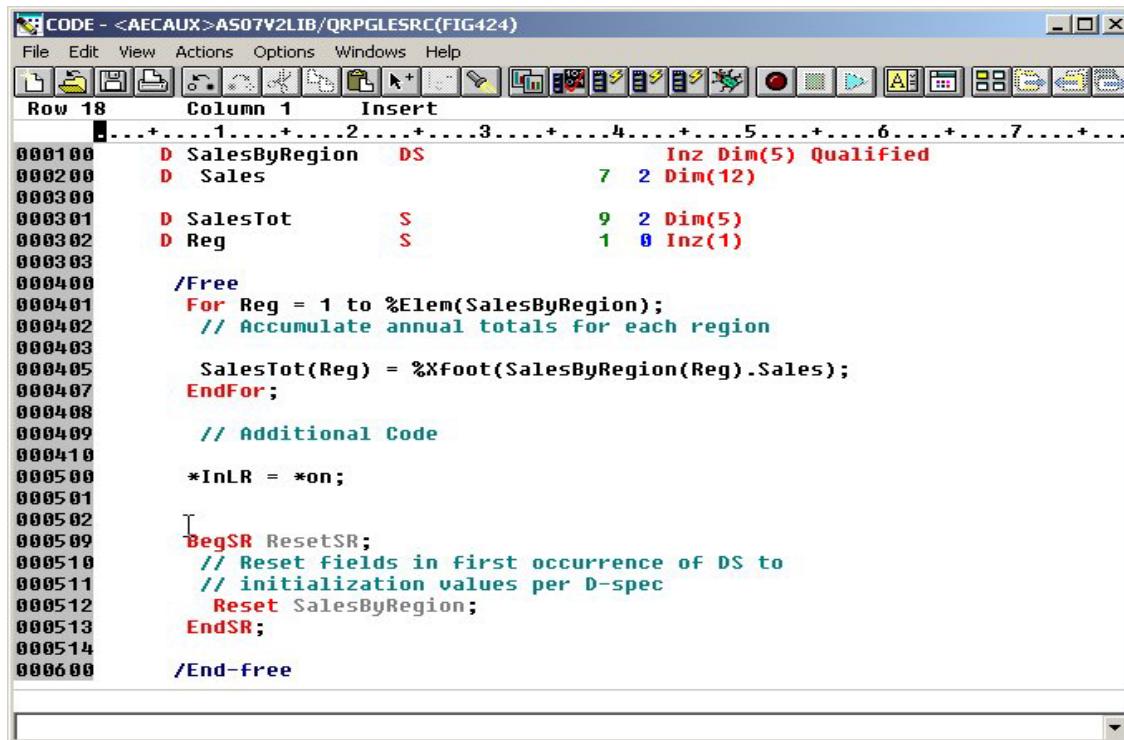
### Notes:

We modify the previous example to use an array data structure. Notice the way we qualify the argument for the **%Xfoot** BIF. We are interested in totals by region and thus want to work with the instance of the **SalesByRegion DS** based on the value of **Reg** as we loop in the **For**.

For multiple occurrence data structures, all occurrences are initialized based upon the data type. You should always use the **INZ** keyword to initialize a data structure and override at the subfield level as required.

# Reinitializing data structures

IBM i



```

CODE - <AECAUX>AS07V2LIB/QRPGLESRC(FIG424)
File Edit View Actions Options Windows Help
Row 18 Column 1 Insert
000100 D SalesByRegion DS           Inz Dim(5) Qualified
000200 D Sales                   7 2 Dim(12)
000300
000301 D SalesTot              S   9 2 Dim(5)
000302 D Reg                    S   1 0 Inz(1)
000303
000400 /Free
000401   For Reg = 1 to %Elem(SalesByRegion);
000402     // Accumulate annual totals for each region
000403
000405     SalesTot(Reg) = %XFoot(SalesByRegion(Reg).Sales);
000407   EndFor;
000408
000409   // Additional Code
000410
000411 *InLR = *on;
000501
000502
000503 BegSR ResetSR;
000504   // Reset Fields in first occurrence of DS to
000505   // initialization values per D-spec
000506   Reset SalesByRegion;
000507 EndSR;
000508
000600 /End-Free

```

© Copyright IBM Corporation 2009

Figure 4-26. Reinitializing data structures

AS075.0

## Notes:

**CLEAR** and **RESET** have many uses but here we show you an example of how **RESET** can now be used to *reinitialize* a data structure, in this case, a multiple occurrence DS.

The **CLEAR** sets elements in a structure to the appropriate default values of blanks for character data and zero for numeric data.

If **\*ALL** is specified, and the result field contains a multiple occurrence data structure or a table name, all occurrences or table elements are cleared, and the occurrence level is set to 1. Note that **\*ALL** is valid with Multiple Occurrence Data Structures only. It is not shown in the visual because its use is not valid. An example of **CLEAR** with **\*ALL**:

```

D SalesByRegion DS           Inz Occurs (5)
D Sales             7 2 Dim(12)

D SalesTot          S   9 2 Dim(5)
D Reg               S   1 0 Inz(1)

/Free

```

```
// Program logic

BegSR ClearDS;
// Clear all fields of DS to default
// initialization values
Clear *All SalesByRegion;
EndSR;
```

**RESET** sets each element in a structure to the initial value that you specified on the D-Spec when you initially set up the Data Structure.

Both these opcodes can also be used with other structures, such as record formats. Review the *RPG IV Reference Manual* for further information.

## What is a data area?

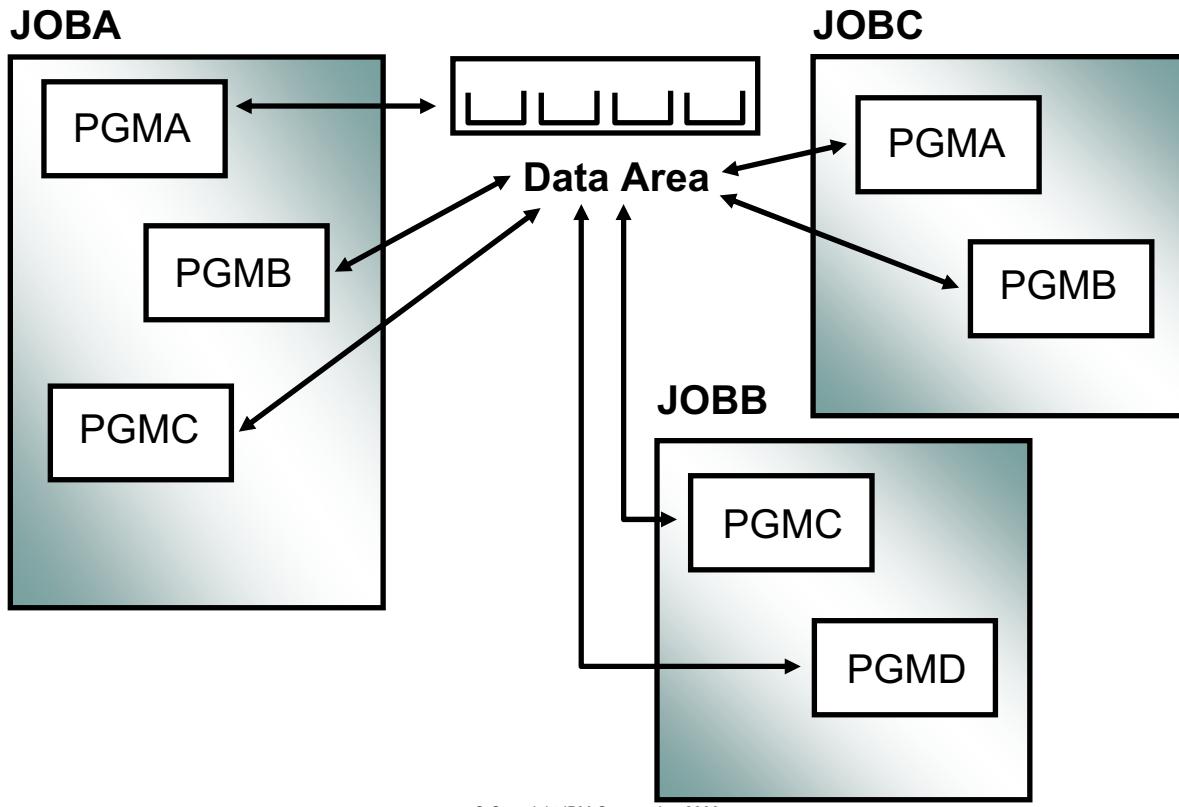


Figure 4-27. What is a data area?

AS075.0

### Notes:

A data area is:

- A permanent object of type \*DTAARA
- Located in auxiliary storage
- Treated by IBM i as a single ‘field’

One type of data area, the *general data area*, is created by the user. It is the only data area that passes data between programs run by different operators, as well as between programs within the same job. A general data area exists until it is explicitly deleted.

A *general data area* is a storage area external to the program. It is accessible by any authorized user. It is a permanent i object, created and deleted using CL commands, and when accessed by a program it is treated by IBM i as a single character string or *field*.

For example:

```
CRTDTAARA DTAARA (AS07V5LIB/AS07DTAARA) TYPE (*CHAR)
LEN(100) VALUE ('RPGDataArea') '
```

## Uses for a data area

IBM i

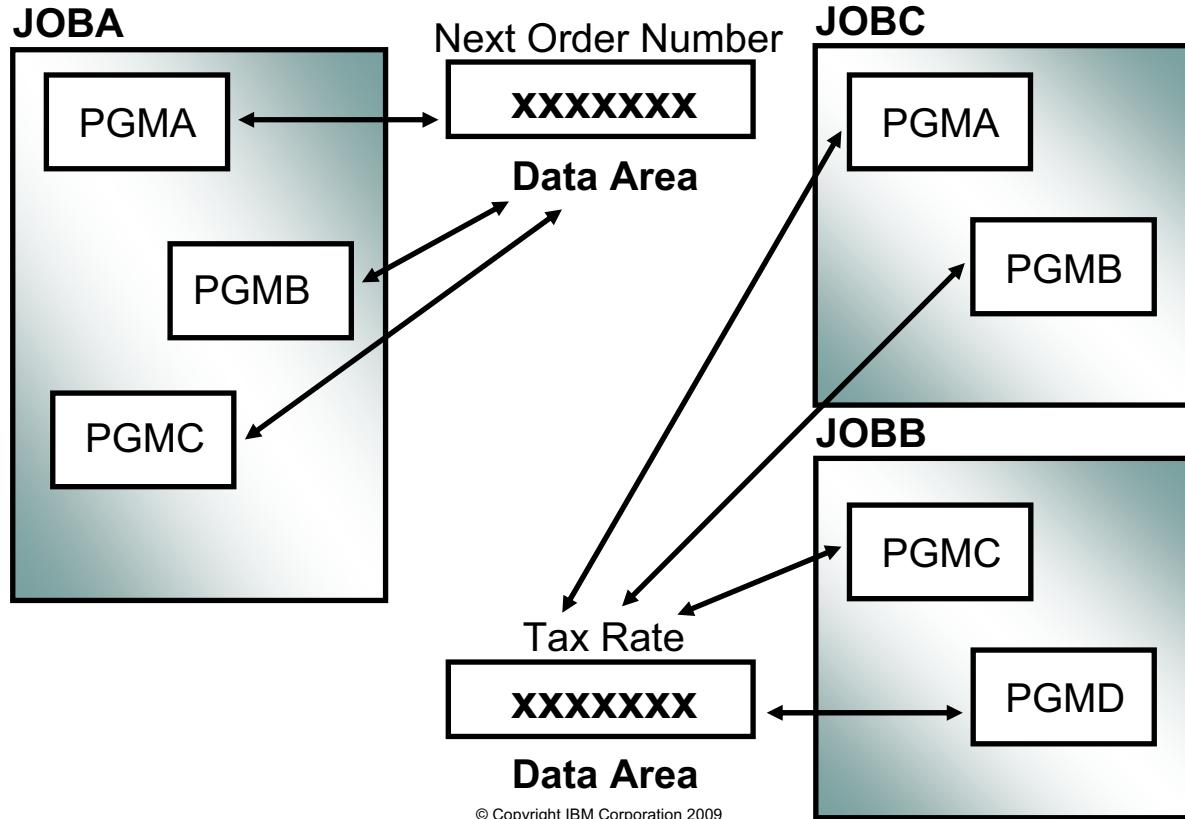


Figure 4-28. Uses for a data area

AS075.0

### Notes:

A data area is used to store control information that is used by multiple jobs. Because there is only one field no cursor is required, so opening a data area does not have the overhead associated with opening a database file. As a result, using a data area for control information is much more efficient than using a *control record* in a file, as might have been done on other systems.

In this example, the **Next Order Number** to be processed and the **Tax Rate** are being stored in data areas.

# Comparison of data area types

IBM i

	General Data Area	Local Data Area	Group Data Area
Created by	User	System	
Size	Variable	Fixed	Only
Type	*CHAR, *DEC, *LGL	*CHAR	Accessed by CL
Referenced	W/I or Between Jobs	W/I Job	
Number Per Job	Variable	One	
Deleted By	User	System	

© Copyright IBM Corporation 2009

Figure 4-29. Comparison of data area types

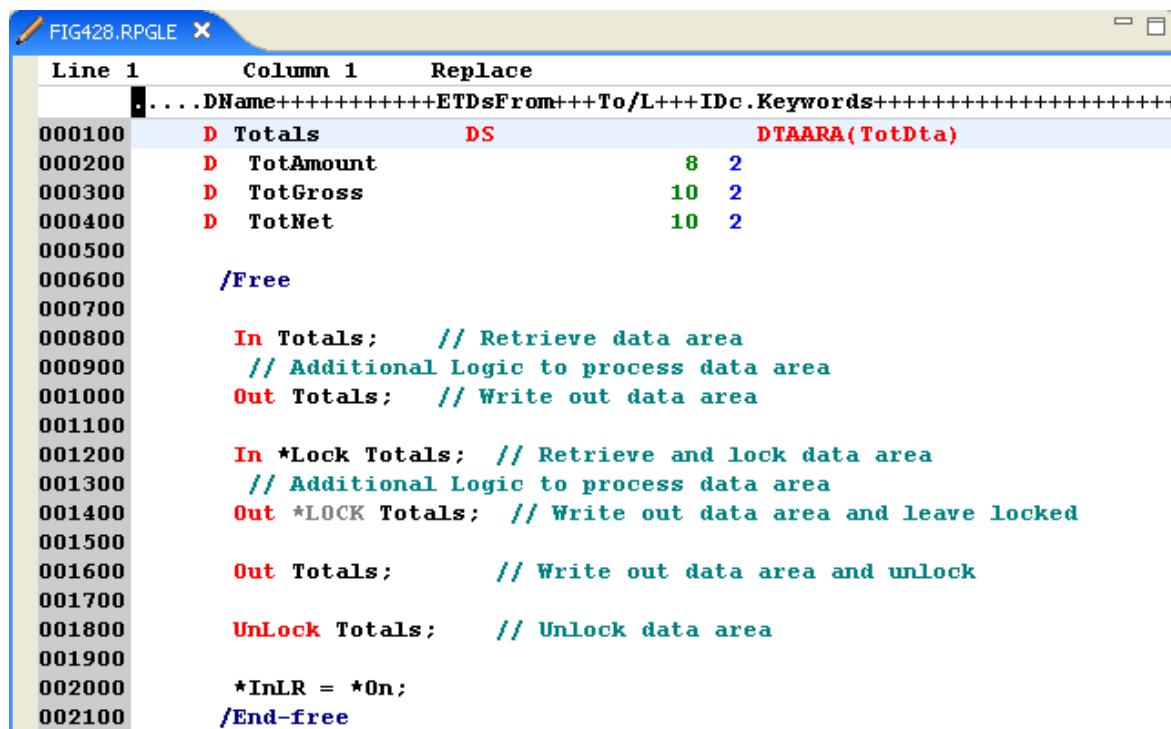
AS075.0

## Notes:

This visual compares the three types of data areas that can be used in a local programming environment. Only a general data area and the local data area (LDA) can be directly accessed by RPG IV.

The maximum length of a general data area is 2000 characters while the \*LDA can be a maximum of 1024 characters.

# Explicitly process a data area



```

Line 1      Column 1      Replace
. .... DName++++++ETDsFrom++To/L+++IDc .Keywords+++++++
000100     D Totals       DS                   DTAARA(TotDta)
000200     D TotAmount    8  2
000300     D TotGross    10  2
000400     D TotNet      10  2
000500
000600     /Free
000700
000800     In Totals;    // Retrieve data area
000900           // Additional Logic to process data area
001000     Out Totals;   // Write out data area
001100
001200     In *Lock Totals; // Retrieve and lock data area
001300           // Additional Logic to process data area
001400     Out *LOCK Totals; // Write out data area and leave locked
001500
001600     Out Totals;    // Write out data area and unlock
001700
001800     UnLock Totals; // Unlock data area
001900
002000     *InLR = *On;
002100     /End-free

```

© Copyright IBM Corporation 2009

Figure 4-30. Explicitly process a data area

AS075.0

## Notes:

To *explicitly* process a data area, a combination of the **IN/OUT/UNLOCK** operations shown on this visual can be used.

You can specify the data area operations (**IN**, **OUT**, and **UNLOCK**) for a data area that is implicitly read in and written out. Before you use a data area data structure with these operations, you would specify that data area data structure name in the D-specs.

When a data area has been defined in a program, its contents can be retrieved by specifying the **IN** operation. If **\*LOCK** is used an exclusive allow read lock state is imposed on the data area and other jobs are prevented from accessing it until the program ends, the data area is read again or an **UNLOCK** operation is used to release it.

The data area can be updated, if necessary, by using the **OUT** operation to write its contents back to disk. The **OUT** operation cannot be used with a data area that has not been locked.

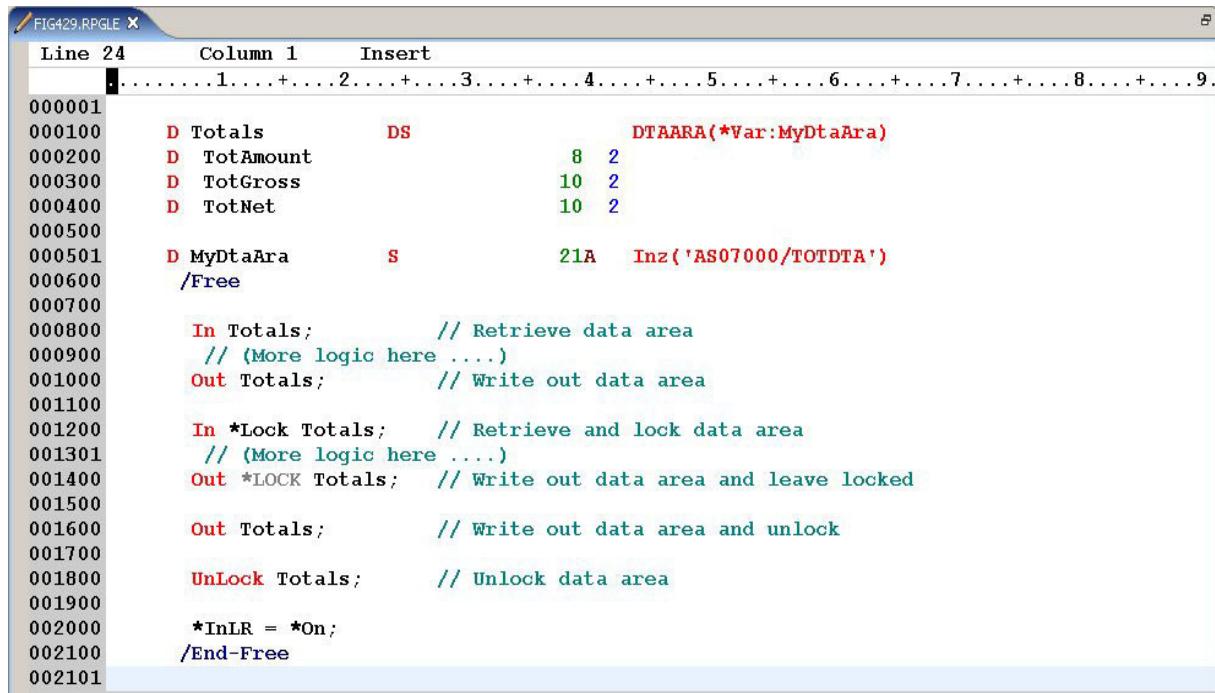
When written with an **OUT**, the data area is unlocked automatically and becomes available for other jobs, unless the **\*LOCK** is specified on the **OUT**. When **\*LOCK** is used in Factor 1

of the **OUT** operation, the data area is locked from updates by other jobs, even after it has been written by this program.

The **UNLOCK** operation can be used to unlock a data area and make it available to other jobs. This operation does not write contents of the data area back to auxiliary storage. All data areas used in the program are unlocked automatically when the program ends. This is *not* true of a sub-program that does not have LR on when its calling program ends. In that case, the data areas remain locked.

# Naming a data area using a variable

IBM i



The screenshot shows an IBM i RPGLE editor window titled 'FIG429.RPGLE'. The code is as follows:

```

Line 24      Column 1      Insert
. .... 1....+....2....+....3....+....4....+....5....+....6....+....7....+....8....+....9.

000001
000100      D Totals        DS          DTAARA(*Var:MyDtaAra)
000200      D TotAmount      8  2
000300      D TotGross       10 2
000400      D TotNet        10 2
000500
000501      D MyDtaAra     S           21A   Inz('AS07000/TOTDTA')
000600      /Free
000700
000800      In Totals;      // Retrieve data area
000900      // (More logic here ....)
001000      Out Totals;     // Write out data area
001100
001200      In *Lock Totals; // Retrieve and lock data area
001301      // (More logic here ....)
001400      Out *LOCK Totals; // Write out data area and leave locked
001500
001600      Out Totals;     // Write out data area and unlock
001700
001800      UnLock Totals; // Unlock data area
001900
002000      *InLR = *On;
002100      /End-Free
002101

```

© Copyright IBM Corporation 2009

Figure 4-31. Naming a data area using a variable

AS075.0

## Notes:

In V5R2, a data area can be named using a program variable. This variable can be supplied from a display format or a passed parameter from another program for example.

The variable must be one of the following forms:

**'NAME'**

**'LIBRARY/NAME'**

**'\*LIBL/NAME'**

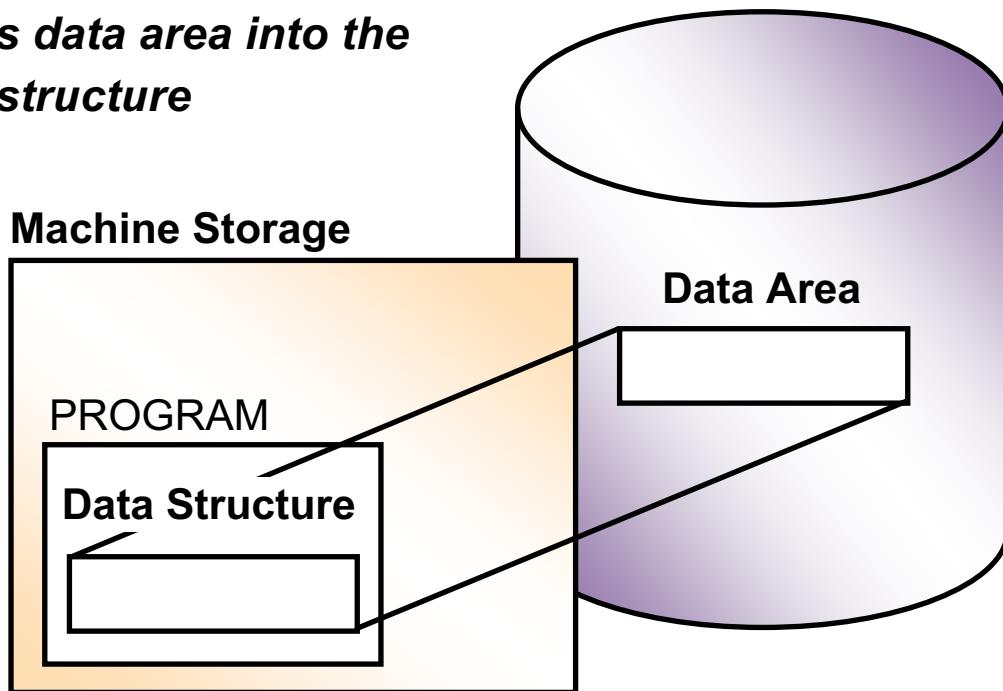
The variable's value must be upper case or the data area will not be found.

In the example, we show the variable as a program variable initialized in the D-specs.

## Implicitly process a data area (1 of 2)

IBM i

**RPG implicitly retrieves and writes data area into the data structure**



© Copyright IBM Corporation 2009

Figure 4-32. Implicitly process a data area (1 of 2)

AS075.0

### Notes:

As we have seen, a data area can be explicitly read and updated by an RPG IV program using **IN** and **OUT**. RPG IV can also use what is known as a data area data structure to implicitly retrieve it during program initialization.

## Implicitly process a data area (2 of 2)

IBM i

```

FIG431.RPGLE X
Line 1      Column 1      Replace
.....DName++++++ETDsFrom+++To/L+++IDc.Keywords+++++
000100    D Totals      UDS
000200    D TotAmount   8 2
000300    D TotGross   10 2
000400    D TotNet    10 2

```

### Data Area Data Structure

© Copyright IBM Corporation 2009

Figure 4-33. Implicitly process a data area (2 of 2)

AS075.0

#### Notes:

In this example, the data area **Totals** might be used to pass information between one program and another. **Totals** is not being accessed by many programs at once, only by perhaps an edit or a processing program. This is an important consideration because the data area will have a shared-no-update lock on it while these programs are running because it is defined as a data area data structure.

The **U** in position 23 on the line that defines the **Totals** data structure indicates that this is a *data area data structure*. The name of the data area, which must be specified in positions 7-21 corresponds with the name of an existing data area. The **U** causes the data area to be implicitly read (and locked) during program activation, and implicitly written out (and unlocked) during program termination.

## Local data area

IBM i

- 1024 bytes of character data
- Created and deleted by IBM i
- \*LDA
- Private to your job
- SBMJOB passes to batch job

---

© Copyright IBM Corporation 2009

Figure 4-34. Local data area

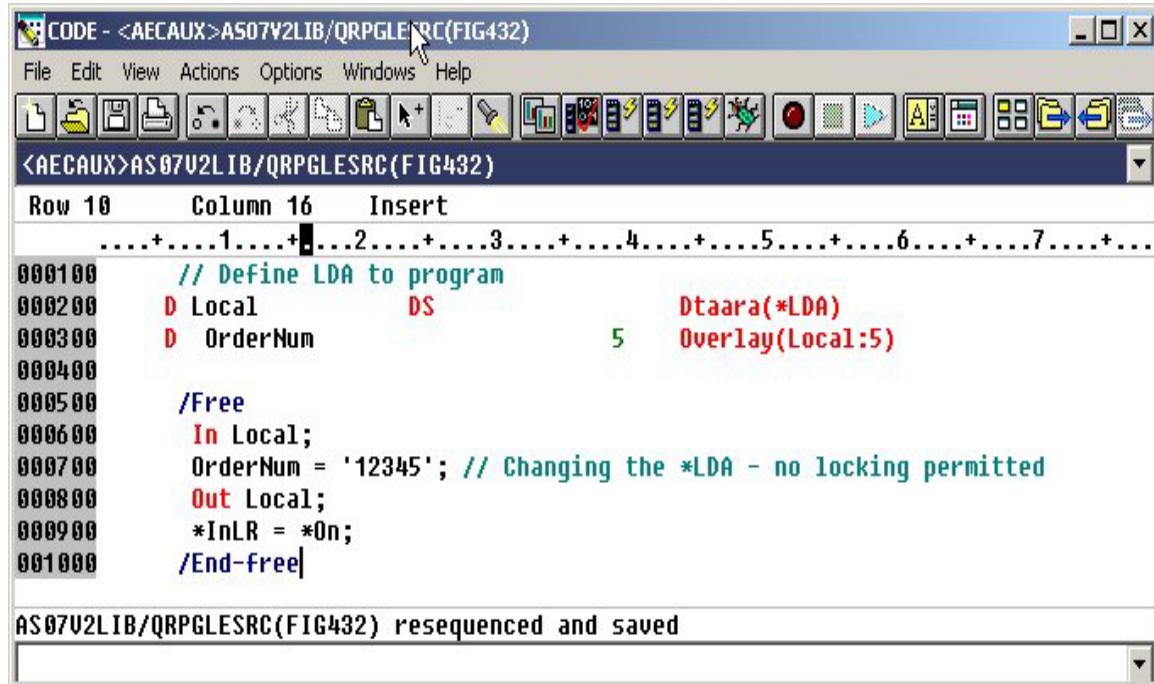
AS075.0

### Notes:

A local data area is a special data area that is automatically created and deleted by the system for each job. The LDA is available, with one exception, only within that job. It is always 1024 bytes of data that can be processed as a character string and is usually referenced as **\*LDA**.

The only case in which the contents of an LDA are available to another job is a SBMJOB that is issued from an interactive job. When the SBMJOB command is run, a snapshot of the current status of the interactive job's LDA is passed to the submitted job and used to set the initial value of its LDA when the batch job is run.

# LDA definition, IN and OUT



The screenshot shows the IBM i CODE editor interface. The title bar reads "CODE - <AECAUX>AS07V2LIB/QRPGLESRC(FIG432)". The menu bar includes File, Edit, View, Actions, Options, Windows, Help. The toolbar contains various icons for file operations. The code window displays assembly language code:

```

<AECAUX>AS07V2LIB/QRPGLESRC(FIG432)
Row 10    Column 16    Insert
.....+....1....+....2....+....3....+....4....+....5....+....6....+....7....+...
000100 // Define LDA to program
000200 D Local          DS
000300 D OrderNum        5      Dtaara(*LDA)
000400
000500 /Free
000600 In Local;
000700 OrderNum = '12345'; // Changing the *LDA - no locking permitted
000800 Out Local;
000900 *InLR = *On;
001000 /End-Free|
AS07V2LIB/QRPGLESRC(FIG432) resequenced and saved

```

© Copyright IBM Corporation 2009

Figure 4-35. LDA definition, IN and OUT

AS075.0

## Notes:

As shown on this visual, there is an advantage to using a data structure for an LDA, even if updates are going to be done explicitly. In this case, the data structure that is being used to process the LDA is named on the D-Specification. Because we referenced a specific subfield of the data structure, positions in the middle of the LDA can be changed.

# Implicit definition of LDA

IBM i

```

FIG434.RPGL X
Line 1      Column 1      Replace
.....1....+....2....+....3....+....4....+....5....+....6....+....7...
000100 // Define LDA to program
000200 D [ ]          UPS
000300 D Local          1024
000400 D OrderNum        5    Overlay(Local:5)

```

LDA implied

© Copyright IBM Corporation 2009

Figure 4-36. Implicit definition of LDA

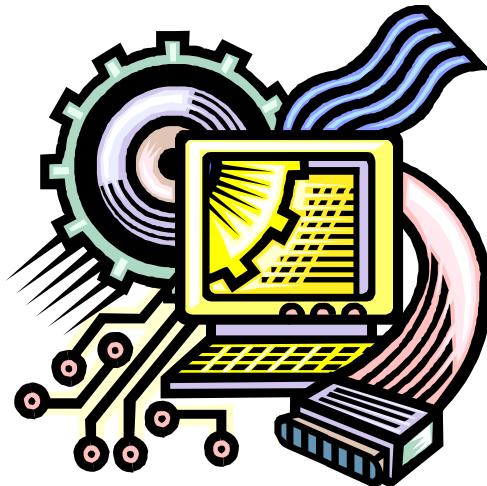
AS075.0

## Notes:

When columns 7 through 21 are blank for the DS name of a data area data structure, the program assumes that the entry refers to the Local Data Area. Otherwise, processing is identical to any other data area using a data structure. Locking is not a concern in this case since the LDA is only available to one job.

# Machine exercise: Data structures and data areas

IBM i



© Copyright IBM Corporation 2009

Figure 4-37. Machine exercise: Data structures/data areas

AS075.0

## Notes:

Perform the machine exercise “Data structures and data areas.”

## Unit summary

IBM i

Having completed this unit, you should be able to:

- Define a data structure in an RPG IV program
- Name two types of data structures and explain how they would be used
- Define a multidimensional array in an RPG IV program
- Use the Like and LikeDS keywords
- Use qualified data structures
- List one use of a data area
- Define a data area in an RPG IV program

© Copyright IBM Corporation 2009

---

Figure 4-38. Unit summary

AS075.0

### Notes:

# Unit 5. Using subfile displays

## What this unit is about

This unit describes the creation and use of subfiles. We cover subfile loading and then accessing subfiles using an inquiry program. Then we discuss more efficient subfile processing varying values of SFLPAG and SFLSIZ. We also discuss how to access and modify subfile records using a relative record number.

## What you should be able to do

After completing this unit, you should be able to:

- Code inquiry programs to display and search a subfile
- Write RPG IV programs to use SFLPAG = SFLSIZ
- Write code to perform page up and page down processing
- Write a subfile maintenance program

## How you will check your progress

- Machine exercises
  - Students write seven subfile programs to practice how to access and maintain subfile data.

# Unit objectives

IBM i

After completing this unit, you should be able to:

- Code inquiry programs to display and search a subfile
- Write RPG IV programs that use SFLSIZ=SFLPAG
- Code page up and page down processing in RPG IV
- Write a subfile maintenance program

© Copyright IBM Corporation 2009

---

Figure 5-1. Unit objectives

AS075.0

## Notes:

## 5.1. Creating a subfile

# What is a subfile?

IBM i

Customer Name Search

Search Code \_\_\_\_\_

Number	Name	Address	City
XXXX	XXXXXXXXXXXXXXXXXXXX	XXXXXXXXXXXXXXXXXXXX	XXXXXXXXXXXXXXXXXXXX
XXXX	XXXXXXXXXXXXXXXXXXXX	XXXXXXXXXXXXXXXXXXXX	XXXXXXXXXXXXXXXXXXXX
XXXX	XXXXXXXXXXXXXXXXXXXX	XXXXXXXXXXXXXXXXXXXX	XXXXXXXXXXXXXXXXXXXX
XXXX	XXXXXXXXXXXXXXXXXXXX	XXXXXXXXXXXXXXXXXXXX	XXXXXXXXXXXXXXXXXXXX
XXXX	XXXXXXXXXXXXXXXXXXXX	XXXXXXXXXXXXXXXXXXXX	XXXXXXXXXXXXXXXXXXXX
XXXX	XXXXXXXXXXXXXXXXXXXX	XXXXXXXXXXXXXXXXXXXX	XXXXXXXXXXXXXXXXXXXX
XXXX	XXXXXXXXXXXXXXXXXXXX	XXXXXXXXXXXXXXXXXXXX	XXXXXXXXXXXXXXXXXXXX
XXXX	XXXXXXXXXXXXXXXXXXXX	XXXXXXXXXXXXXXXXXXXX	XXXXXXXXXXXXXXXXXXXX
XXXX	XXXXXXXXXXXXXXXXXXXX	XXXXXXXXXXXXXXXXXXXX	XXXXXXXXXXXXXXXXXXXX
XXXX	XXXXXXXXXXXXXXXXXXXX	XXXXXXXXXXXXXXXXXXXX	XXXXXXXXXXXXXXXXXXXX
XXXX	XXXXXXXXXXXXXXXXXXXX	XXXXXXXXXXXXXXXXXXXX	XXXXXXXXXXXXXXXXXXXX
XXXX	XXXXXXXXXXXXXXXXXXXX	XXXXXXXXXXXXXXXXXXXX	XXXXXXXXXXXXXXXXXXXX
XXXX	XXXXXXXXXXXXXXXXXXXX	XXXXXXXXXXXXXXXXXXXX	XXXXXXXXXXXXXXXXXXXX
:	:	:	:

© Copyright IBM Corporation 2009

Figure 5-2. What is a subfile?

AS075.0

## Notes:

A *subfile* is a group of records that are identical in format that is read from or written to a display file.

The program reads a group of records from a database file and then creates a subfile by writing records to the subfile that are based on those database records.

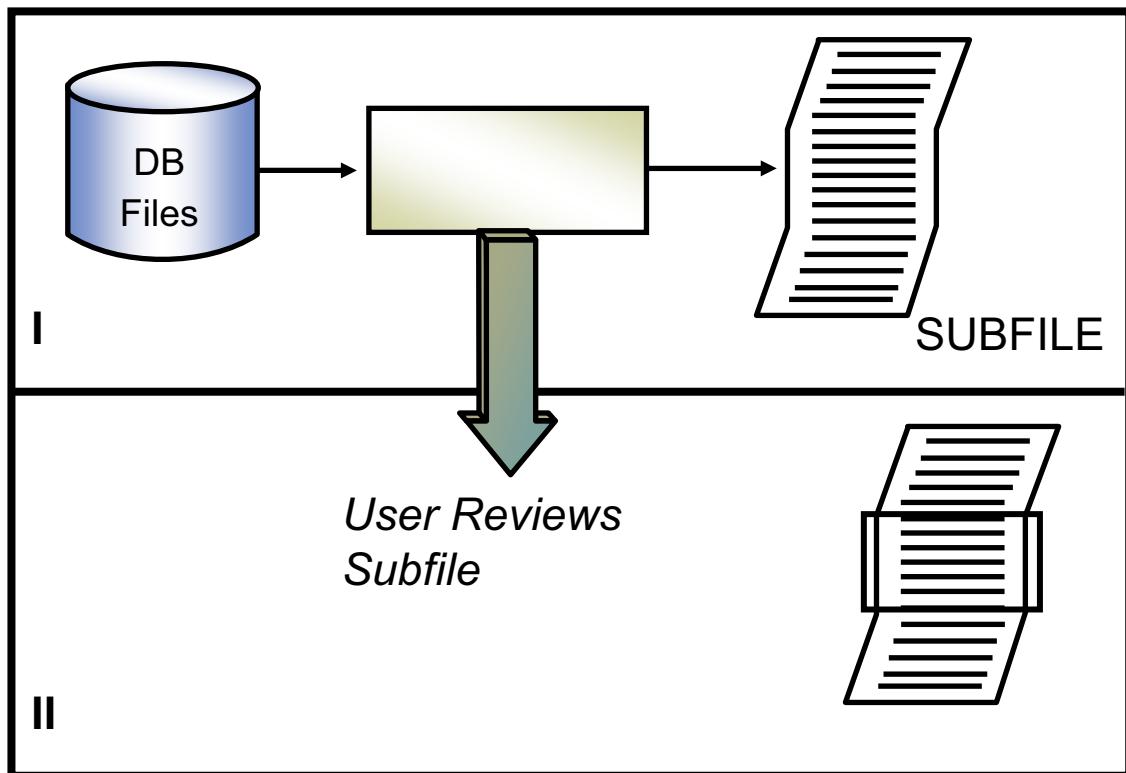
When the entire subfile has been written, the program displays the subfile on the display device in one write operation. Because the subfile can contain more records than can be physically displayed, DDS provides a facility to allow automatic scrolling that you do not have to manage in your RPG IV code.

In an update application the workstation user would change data or enter additional data in the subfile. The program would then read the entire subfile from the display device into the program and process each record from the subfile individually.

Subfiles can be specified in the DDS for a display-device file to allow you to handle multiple records of the same type on the display.

## Subfile used for inquiry

IBM i



© Copyright IBM Corporation 2009

Figure 5-3. Subfile used for inquiry

AS075.0

### Notes:

A subfile is a *temporary storage area* that is *associated with a display file*. It can have records written to it or read from it. Its purpose is to facilitate the presentation of *multiple records of identical format* at one time on a display. Subfiles can be used for data entry, record update and inquiry programs that allow a user to interface with multiple records of a common format.

This visual shows the two stages of processing required when using a subfile:

1. The RPG IV program writes records to the subfile. The subfile exists in memory or on disk. Records are written to it in a fashion similar to the way that records are written to a physical file. The program writes a single record at a time to the subfile, using the subfile's relative record number to determine where the record will be placed. Each record requires a WRITE operation in the RPG IV program.
2. A subfile can be created with a number of records that exceeds the number of lines available on the physical display. When this is done, the system manages the page up, page down or the roll forward, roll backward keys so that the user can scroll up and down the complete subfile.

The programmer decides how large the subfile should be. *A subfile can be extended!* While the program runs, it writes records to the subfile until one of two things happens:

1. The program satisfies the retrieval request for a specific number of records, or all records that meet a specific condition, and writes those records in the subfile.
2. The program writes enough records to fill or partially fill the subfile.

When either condition exists, the program stops writing records and then issues an operation that displays the first page of records to the user. The user can now scroll through multiple pages of the subfile, reviewing the records that are in the subfile. The user can then enter criteria for another search or request end of job.

As you examine the two stages of subfile use on this visual, you see that:

1. The program first writes records to the subfile.
2. The program writes records from the subfile to the display.

# Sample inquiry subfile application

IBM i

05/11/00		ITEM LIST		RJSLANEY
Item No	Description	Qty on Hand	Qty on Order	
20001	Telephone, one line	10	6	
20002	Telephone, two line	5	12	
20003	Speaker Telephone	6	8	
20004	Telephone Extension Cord	25	10	
20005	Dry Erase Marker Packs	428	10	
20006	Executive Chairs	10	2	
20007	Secretarial Chairs	13	5	
20008	Desk Calendar Pads	56	10	
20009	Diskette Mailers	128	200	
20010	Address Books	1,680	0	
20011	Desk lamp, brass	3	24	
20012	Blue pens	7	10	
20013	Red pens	14	5	
20014	Black pens	25	12	
20015	Number 2 pencils	150	20	
				More...

Press Enter to go to top of file  
F3=Exit

© Copyright IBM Corporation 2009

Figure 5-4. Sample inquiry subfile application

AS075.0

## Notes:

This is the output of a sample application that is implemented using a subfile. We have an inventory application and we simply want to display the items in inventory and allow the user to scroll through the list until F3 is pressed.

# Sample inquiry subfile application: DDS

IBM i

```

000100 A
000200 A
000201 A
000300
000400 ** Subfile Record Format
000500>>1 A R RECORD
000600 A ITMNR R 0 5 7
000700 A ITMDESCR R 0 5 20
000800 A ITMQTYOH R 0 5 52EDTCDE(1)
000900 A ITMQTYOO R 0 5 66EDTCDE(1)
001000
001100 ** Subfile Control Format
001200>>2 A R CONTROL
001300>>3 A
001400>>4 A
001500>>5 A
001600>>6 A
001700>>7 A 98
001800>>8 A
001900
002000 A R HEADING
002100 A
002200 A
002300 A
002400 A
002500 A
002600 A
002700 A
002800
002900 ** Function key narrative
003000 A R FNKEYS
003100 A
003200 A
003300 A

REF(ITEM_PF)
CA83(83)
INDARA

SFL<-->1
SFLCTL<-->2
SFLDSP<-->3
SFLSIZ(0016)<-->4
SFLPAG(0015)<-->5
SFLEND(*MORE)<-->6
OVERLAY<-->7
OVERLAY<-->8

2 7DATE EDTCDE(Y)
2 36'ITEM LIST' DSPATR(HI)
2 65USER
4 7'Item No'
4 20'Description'
4 50'Qty on Hand'
4 63'Qty on Order'

21 7'Press Enter to go to top of file'
COLOR(BLU)
22 7'F3=Exit'

```

© Copyright IBM Corporation 2009

Figure 5-5. Sample inquiry subfile application: DDS

AS075.0

## Notes:

Much of this DDS looks similar to the DDS from previous examples and exercises. We focus our attention on the differences that are noted in the visual. Notice the subfile data record and the subfile control record specifically. The other records and OVERLAY are used as they were in simple inquiries. We have been emphasizing modularity in your display design. It is common practice to include the HEADING and CONTROL formats in a single subfile control format. You see this used in many display files.

Notice that the subfile data record and subfile control record must be coded in the sequence shown and must be coded together. If you order them in any other way, you get a compile error when you try to create your DDS.

Listed in the visual are commonly used keywords for subfiles. Four of them are required in addition to SFL and SFLCTL, regardless of the intended use for the subfile. SFLSIZ, SFLPAG, SFLDSP, and SFLDSPCTL are *mandatory* keywords. They must be coded just to get the DDS display file source to compile when using subfiles:

1. **SFL** identifies this as a subfile record format. This record format including its related field descriptions must immediately precede the subfile control record format identified by the **SFLCTL** keyword.
2. **SFLCTL** record-level keyword is used to specify that this record format is a subfile control record format. This record format must immediately follow the subfile record format.

The format of the keyword is:

**SFLCTL (subfile-record-format-name)**

You must specify the name of the subfile record format as the parameter value for this keyword. The subfile control record format can contain field descriptions as well as subfile control keywords.

The subfile record format (SFL keyword) defines the format of the record in the subfile as opposed to the subfile control record format (SFLCTL keyword), which defines how the subfile can be displayed, cleared, deleted and initialized. The program sends output operations to the subfile record format to build the subfile. It also sends output operations to the subfile control record format, setting option indicators for various subfile keywords to display, clear, delete and initialize the subfile.

3. **SFLDSPCTL** tells the system to display fields coded in the control record format when the program performs an output operation to the subfile control record format. This includes all constants as well as any fields. An indicator is often associated with this keyword in order to control completely what information is displayed for the subfile and when.
4. **SFLDSP** tells the system when to display a page of subfile records from the subfile. An option indicator is often used to determine when a page of records will be displayed. If no indicator is used, the system attempts to show a page of subfile records every time the program performs an output operation to the subfile control record format.



### Note

#### SFLDSPCTL/SFLDSP

SFLDSPCTL and SFLDSP together are used to display the complete display (that is, headings, fields, subfile records) to the user. SFLDSP cannot be performed without SFLDSPCTL.

5. **SFLSIZ** is used to specify the total number of records in the subfile. A subfile can have a maximum of 9,999 records. When an RPG program writes a record that fills a subfile, the system returns a message to the program. You can also use the %eof BIF is used to retrieve the file full message.

Most programs limit the number of records included in a subfile because the user does not want to wait while the subfile is being filled. Another reason for limiting the number

of subfile records is the possible need to lock these records from other users' updates while the program is using the records in its subfile.

6. **SFLPAG** is used to indicate the number of records in the subfile that are to be displayed per 'page'.
7. **SFLEND** controls display of a plus sign (or in this case, *More...*) in the lower right hand corner of the subfile display indicating that there are more records to be displayed. It is used with an option indicator to manipulate the appearance of the plus (+) sign (or *More....*) on the display. If SFLPAG does not equal SFLSIZ, the indicator that is set on when the search is completed should be the one that is used to control the display of the plus sign.

When the indicator is off, the + (or *More...*) is always displayed. If the indicator is on, the + (or *More...*) is shown if there are more records to display and blank (or *Bottom*) if there are no more records to display.

8. **OVERLAY** is used to specify that the record format you are defining should appear on the display without the entire display being cleared first.

Normally, the entire display is deleted on each output operation. All records on the display with fields that partially or completely overlap fields in this record are deleted before this record is displayed; all others remain on the display and are not changed in any way. A record already on the display is deleted even if fields specified in the record format are not selected for display.

# Sample inquiry subfile application: Program

IBM i

```

000001 . . . . . FFilename++ IPEASFRlen+LKlen+AIDevice+ .Keywords+++++
000002   FItem_PF  IF   E          K Disk
000003   FItemDsp  CF   E          Workstn SFile(Record:Rrn)
000004   F          IndDS(WkStnIndics)
000005   D WkStnIndics    DS
000006   D Exit           3      3N
000007   D SF1End         98     98N
000008
000009   D Rrn            S          4   0
000010
000011 /FREE
000012>> 1 Write FnKeys; // Write FnKeys and
000013   Write Heading; // Heading Formats
000014   Read Item_PF; // Read First record of Item_PF file
000015   Rrn = 1;
000016
000017>> 2 // Load subfile with data
000018   Dow Not %eof(Item_PF) and Not %eof(ItemDsp);
000019     Write Record;
000020     Read Item_PF;
000021     Rrn = Rrn +1;
000022   Enddo;
000023
000024   SF1End = *on; // Set SF1END now that subfile loaded
000025
000026>> 3 Dow Not Exit;
000027   ExFmt Control; // Display subfile
000028 Enddo;
000029
000030   *InLR = *On;
000031 /END-FREE

```

© Copyright IBM Corporation 2009

Figure 5-6. Sample inquiry subfile application: Program

AS075.0

## Notes:

This is the program that produces the subfile inquiry display shown earlier (using the DSPPF shown in the previous visual).

This program loads the subfile the data records from the inventory file, and, then, displays the subfile such that the user can scroll up and down or end the program by pressing F3. In the highlighted steps in the source code above:

1. The **WRITE** opcode moves the constant data (in the **FNKEYS** and **HEADING** formats) to the display buffer where it is held pending an **EXFMT** operation, that performs the physical write to the display.

Because the field names in the Display DDS are identical to those of the externally described **Item\_PF** data file, it is unnecessary to code any expressions to assign program data to the subfile record. The program reads the first data record of the **Item\_PF** file and initializes the value of the **RRN** to 1.

2. The **DoW loop** is executed until End-of-File is reached in the **Item\_PF** file. Each record read is written to the subfile using the subfile data record format until we reach EOF of **Item\_PF**.

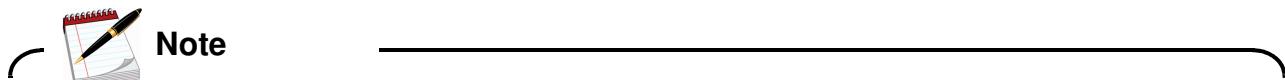
Indicator **SflEnd(Mapped to \*in90)** is set on when we are ready to display the subfile. Either end-of-file of **Item\_PF** has occurred or we have reached end-of-file of the subfile.

The **RRN** will be used to set the file cursor to the next record in the subfile as we continue to **READ** the records in the **Item\_PF** file. When end-of-file is reached, we have filled the subfile and are ready to display it.

3. Finally, we display the subfile using a **DoW** loop that will be executed until the user presses F3. Note that we **EXFMT** the **CONTROL** record of the subfile.

**SFLEND(Mapped to \*in90)** is on at this point and is also used in the display file to display **More.../Bottom** in the lower right hand corner to indicate that there are either more records (**More...**) to view or no more records (**Bottom**) to view.

When F3 is pressed, we exit the application.



### Additional Coding Required

To make this program complete, we would need to handle the possibility that the subfile is filled before all the data records have been read. To keep the program simple, we did not code this possibility. Also, notice that we could have coded a single test for end of file using the **%eof** BIF. However, to make what we are doing perfectly clear, we coded two specific tests.

## Writing records to a subfile

IBM i

**Use subfile record format to load data into subfile.**

**WRITE**

*Relative Record  
Number  
Will Determine Where  
Record Written*

© Copyright IBM Corporation 2009

Figure 5-7. Writing records to a subfile

AS075.0

### **Notes:**

The **WRITE** operation is used with the *subfile data record format* to write records in the subfile. A relative record number determines where the record will be placed.

The subfile record format contains the field information that is transferred to or from the display file under control of the subfile control-record format.

# Writing records to a subfile: RPG IV

IBM i

```

Open. 1.....2.....3.....4.....5.....6.....7.....+
100001 FItem_PF IF E K Disk
100002>>1 FItemDsp CF E Workstn SFile(Record:Rrn)
100003 F IndDS(WkStnIndices)
100004
100005 D WkStnIndices DS
100006 D Exit 3 3N
100007 D SF1End 98 98N
100008
100009 D Rrn S 4 8
100010
100011 //FREE
100012 Write FnKeys; // Write FnKeys and
100013 Write Heading; // Heading formats
100014 Read Item_PF; // Read first record of Item_PF file
100015 Rrn = 1;
100016
100017 // Load subfile with data
100018>>3 Dov Not %eof(Item_PF) and Not %eof(ItemDsp);
100019 Write Record;
100020 Read Item_PF;
100021 Rrn = Rrn +1;
100022 Enddo;
100023
100024 SF1End = *on; // Set SF1END now that subfile loaded
100025
100026 Dov Not Exit;
100027 ExFmt Control; // Display subfile
100028 Enddo;
100029
100030 *InLR = *On;
100031 /END-FREE

```

© Copyright IBM Corporation 2009

Figure 5-8. Writing records to a subfile: RPG IV

AS075.0

## Notes:

1. The F-spec for **ItemDsp** defines the display file as using a program variable (RRN) that is used as the cursor to process subfile records. This field is used as a relative record number counter to indicate where to write the next record. The program is responsible for incrementing this number prior to writing the record to the subfile. The value of this field should never be negative or greater than the number of records that can be contained in the file. A function check occurs if an attempt is made to write a record and the RRN is zero or too large.
2. Each time a record is written to the subfile, the relative record number counter is incremented to determine where the next record is added.
3. When the number of records in the subfile equals the value of the keyword SFLSIZ, we reach the end of file in the subfile. When we reach end of file in the subfile, we exit the loop. We test for end of file of the subfile using a DoW. Also, we could reach end of file of the Item\_PF file so we test for this condition as well. Obviously, if we have reached end of data, the subfile is as full as we can make it.

**Note**

If **SFLSIZ** is greater than **SFLPAG** in the subfile control format, the program might continue to add records to the subfile even after the subfile full indicator has been turned on. An additional extent equal to the SFLSIZ parameter is added to the subfile. When this extent is filled, the subfile full indicator is set on, again.

If **SFSIZ = SFLPAG**, the SFLSIZ is not increased and you are responsible to handle scrolling in your program.

# Performing display I/O and control functions

IBM i

***Use Control Record to Display  
Subfile  
or Perform Control Functions***

**WRITE  
EXFMT**

***Set indicators to control  
what actions are  
performed***

© Copyright IBM Corporation 2009

Figure 5-9. Performing display I/O and control functions

AS075.0

## Notes:

The DDS for a subfile consists of at least these two record formats:

### 1. Subfile record format

The subfile record format contains the field information that is transferred to or from the display file under control of the subfile control record format.

### 2. Subfile control-record format

The subfile control-record format causes the physical read, write or control operations of a subfile to take place. What actions take place are determined by the keywords you code and indicator settings.

# Physical display I/O: EXFMT

IBM i

```

-----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+
002200  AAN01N02N03 ..... Functions+++++
002201
002300  ** Subfile Control Format
002400>1 A R CONTROL           SFLCTL(RECORD)
002500>2 A 85                SFLDSPCTL
002600>3 A 95                SFLDSP
002700    A 75                SFLCLR
002800    A SFLSIZ(0011)
002900>4 A SFLPAG(0010)
003000    A SFLEND(*MORE)
                                OVERLAY

-----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+
000001  FItem_PF  IF   E          K Disk
000003  FItemSubs1 CF   E        Workstn Sfile(Record:Rrn)
000004  F                           IndDS(WkStnIndices)
000005
000006  D WkStnIndices      DS
000007  D Exit               3     3N
000008  D SF1End             90    90N
000009  D SF1DspCtl          85    85N
000010  D SF1Dsp              95    95N
000011  D SF1Clr              75    75N
000012
000013  D Rrn                S      4  0 INZ
000014
000015  /Free
000016>>1 SF1DspCtl = *On; // Display Subfile
000018>>2 SF1Dsp = *On;
000019
000020  ExFmt Control;
000021
000022>>1 SF1DspCtl = *Off; // Reset indicators
000023>>2 SF1Dsp = *Off;
000024  // More logic
000025  /End-free

```

© Copyright IBM Corporation 2009

Figure 5-10. Physical display I/O: EXFMT

AS075.0

## Notes:

In the DDS, we specify indicators to determine whether certain functions are performed. For example, in the DDS above, we code indicators that determine whether the subfile record format is displayed, whether the subfile is cleared using the **SFLCLR** keyword. This record-level keyword is used in the subfile control record format so that your program can clear the subfile of all records. Clearing the subfile does not affect the display. However, after being cleared, the subfile contains no active records.

To clear the subfile, the RPG program would turn **SfIClIr (Mapped to \*in75)** on. An option indicator should be used with this keyword to prevent clearing of the subfile on every output operation.

Further notes:

1. **SfICtlDsp (Mapped to \*in85)** controls the display of the control record.
2. **SfIDsp (Mapped to \*in95)** is tied to the display of the subfile records. Thus, indicators SfICtlDsp and SfIDsp (Mapped to indicators 85 and 95) together instruct the system to display the control format and the subfile records.

3. **SfIClr (Mapped to \*in75)**, as noted, can be used to clear the subfile of all records.

When records already exist in the subfile and all records are to be replaced, you can use SfICrl (Mapped to \*in75) to clear the subfile, followed by an output operation to the subfile control record format using **SfICtlDsp** and **SfIDsp**. Remember that the **SFLCLR** keyword clears the subfile (**not the display**) and permits the program to write new records to the subfile.

In using the program, and indicators, to control clearing the subfile and the display of the subfile and the subfile control record, the end user never sees the screen being cleared and refreshed.

4. **SfIEnd (Mapped to \*in90)** is used to condition the displaying of More... / Bottom on the screen.

EXFMT writes the format specified and then waits for input from the workstation. Because indicators **SfICtlDsp** and **SfIDsp** have been set on, this operation is used to display the subfile contents and the heading and control information in the subfile record control format.

Either the Enter key or a special function key must be pressed to return control to the program.

## Control function example: Clear subfile

IBM i

```

000001  FItem_PF  IF  E          K Disk
000002  FItemSubs1 CF  E          Workstn Sfile(Record:Rrn)
000003  F                      IndDS(WkStnIndices)
000004
000005  D WkStnIndices    DS
000006  D Exit             3     3N
000007  D SF1End           90    90N
000009  D SF1DspCtl        85    85N
000010  D SF1Dsp           95    95N
000011  D SF1Clr            75    75N
000012
000013  D Rrn               S     4  0 INZ
000013----- 60 data records excluded -----
000120  Begsr SF1Clear; // Subfile clear subroutine
000121      SF1Clr = *on;
000122      SF1Dsp = *OFF;
000124      SF1DspCtl = *OFF;
000126      Write Control; // New search - clear subfile
000127      SF1Clr = *off;
000129  EndSR;

```

© Copyright IBM Corporation 2009

Figure 5-11. Control function example: Clear subfile

AS075.0

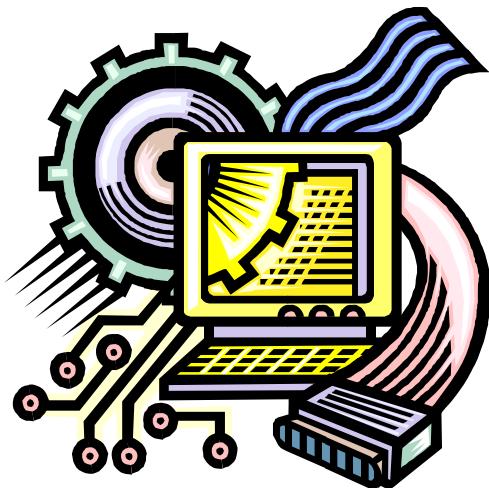
### Notes:

The **WRITE** operation is used with the subfile control record format when the program needs to clear the subfile. This could be a user-controlled function invoked by the user pressing F5 (where F5 = Refresh) or it could be done when the user specifies a new search argument used to position the cursor in the subfile.

Because **SFLDSP** and **SFLDSPCTL** have been disabled, the user does not see the subfile being cleared and the display remains as it was. The program would need to perform an output operation (**ExFmt**) to the display that would refresh the data that is displayed.

## Machine exercise: Inquiry subfiles

IBM i



© Copyright IBM Corporation 2009

Figure 5-12. Machine exercise: Inquiry subfiles

AS075.0

### Notes:

Perform the machine exercise “Inquiry subfiles.”

## 5.2. Adding a search argument

# Search argument with subfile: Output

IBM i

6/23/00                    Item List Search                    RJSLANEY

Enter item number: [20015](#)

Press Enter to continue or to go to top of file

Item No	Description	Qty on Hand	Qty on Order
20015	Number 2 pencils	150	20
20016	Number 3 pencils	25	10
20017	Two Drawer File Cabinets	15	6
20018	Manilla folders	50	5
20019	Hanging file folders	150	12
20020	Metal desk	2	0
20021	Pink erasers	250	10
20022	Gum erasers	100	5
20023	Twelve inch rulers	15	10
20024	Eighteen inch rulers	65	5
20025	Staples	14	40
20026	Three Ring Binders	10	108

More...

F3=Exit

© Copyright IBM Corporation 2009

Figure 5-13. Search argument with subfile: Output

AS075.0

## Notes:

In the subfile inquiry example, we saw that because we displayed the complete subfile, it was necessary for the user to scroll up and down to search visually for a specific record. This is acceptable when the database file contains only 50 records as is the case of our **ITEM\_PF** database file. But users would very quickly become dissatisfied if they had to scroll through thousands of records time after time to get to the records they needed to see.

There is a solution. It is to add a search argument to the program that can be used to position the file cursor. When the file cursor has been set, we could use READ operations to extract all records in the database from that point forward or we could even further limit the scope of the search.

This visual shows the use of a search argument that is the item number key to the file.

In this example, the user entered a search argument value of **20015**. Our program then reads the specific record with this key and displays it and all those that follow from that point forward to the end of file of the **Item\_PF** database file.

The user can work with the desired record and then reset the search argument to reposition the file cursor and start again. By pressing the Enter key without entering a search argument, the file cursor is set at the beginning of file.

If the user were to enter a search argument that positioned the cursor at end of file, a message might be displayed.

# Search argument with subfile: DDS

IBM i

```

000100      A
000200      A
000201      A
000300
000400>> 1 A          R PROMPT
000500
000600
000700
000800
000900
001000      A          ITMSCH   R     D   I
001100      A          96
001200
001300
001400
001500
001600>> 3 A          R RECORD
001700      A          ITMNBR   R     0  9  7
001800      A          ITMDESCR R     0  9  28
001900      A          ITMQTYOH R     0  9  52EDTCDE(1)
002000      A          ITMQTYOO R     0  9  66EDTCDE(1)
002100
002200      ** Subfile Record Format
002300      A          R CONTROL
002400      A          85
002500      A          95
002600      A          75
002700
002800>> 2 A          R HEADING
002900      A          98
003000
003100
003200      A          R FNKEYS
003300
003400
003500
003600
003700
003800>> 3 A          R OVERLAY
003900

REF(ITEM_PF)
CA03(03)
INDARA

OVERLAY
2 7DATE EDTCDE(Y),
2 32'Item List Search',
DSPATR(HI)
2 65USER
3 2'Enter item number:'
3 31REFFLD(ITMNBR)
ERRMSG('No Items found' 96)
4 7'Press Enter to continue or to go -
to top of file'
COLOR(BLU)

SFL
SFLCTL(RECORD)
SFLDSPCTL
SFLDSP
SFLCLR
SFLSIZ(0016)
SFLPAG(0012)
SFLEND(*MORE)
OVERLAY

OVERLAY
7 7'Item No'
7 20'Description'
7 50'Qty on Hand'
7 63'Qty on Order'

23 7'F3=Exit'

```

© Copyright IBM Corporation 2009

Figure 5-14. Search argument with subfile: DDS

AS075.0

## Notes:

To add a search argument to the subfile inquiry application, we first need to specify it in our DSPF DDS. We use the ITEMDSP display file that we showed you earlier as a basis and modify it.

We add an input field, a comment to direct the user on what to do and an error message to indicate a problem (the message `No items found` means that the subfile is empty).

In addition we made the following minor changes:

1. Modified and moved some information from the HEADING to the PROMPT format to add prompting of the search argument (ensuring that formats do not overlap one another as well).
2. Decreased the value of SFLPAG from 15 records to 12 so that no formats would be overlaid.
3. Altered the line numbers for the display of the subfile and the FNKEYS to avoid possible overlay.

The main effort in this process was adding the search argument. All other changes are cosmetic in nature. Remember, you would expect to change the screen design if you were adding more data and formats to the display file, as we are in this case.

# Search argument with subfile: Flowchart (1 of 3)

IBM i

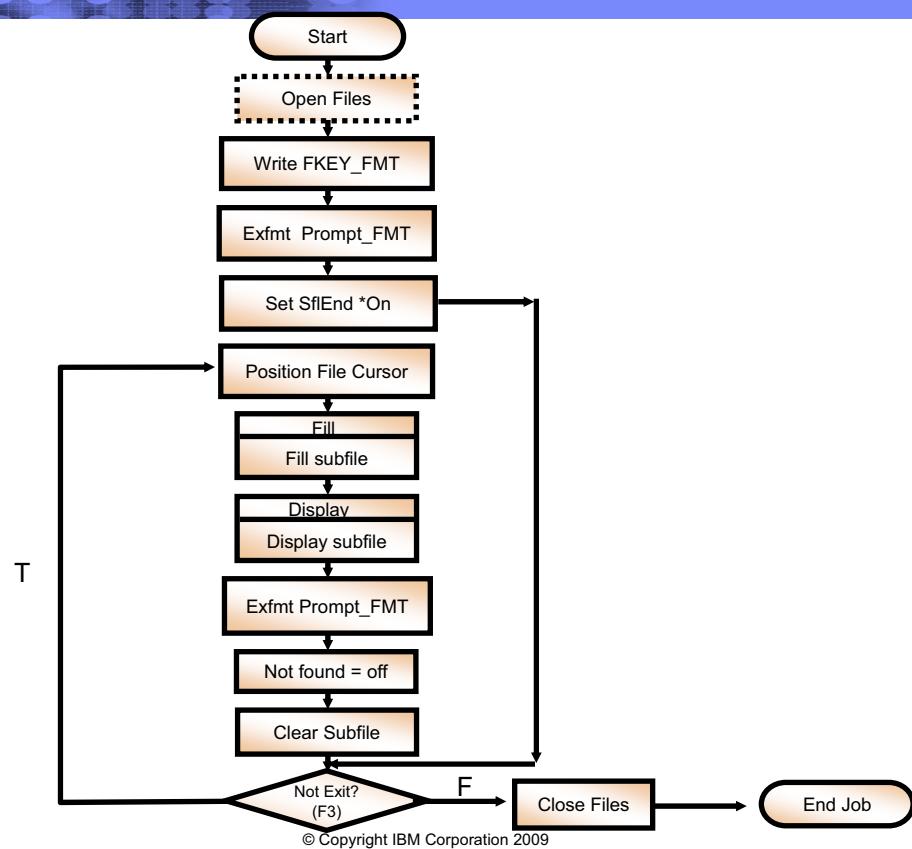


Figure 5-15. Search argument with subfile: Flowchart (1 of 3)

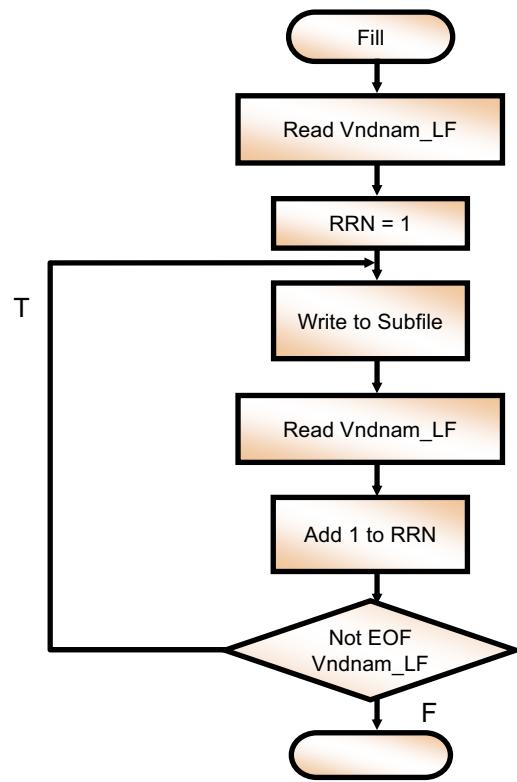
AS075.0

## Notes:

Use these charts when you code the machine exercise.

## Search argument with subfile: Flowchart (2 of 3)

IBM i



© Copyright IBM Corporation 2009

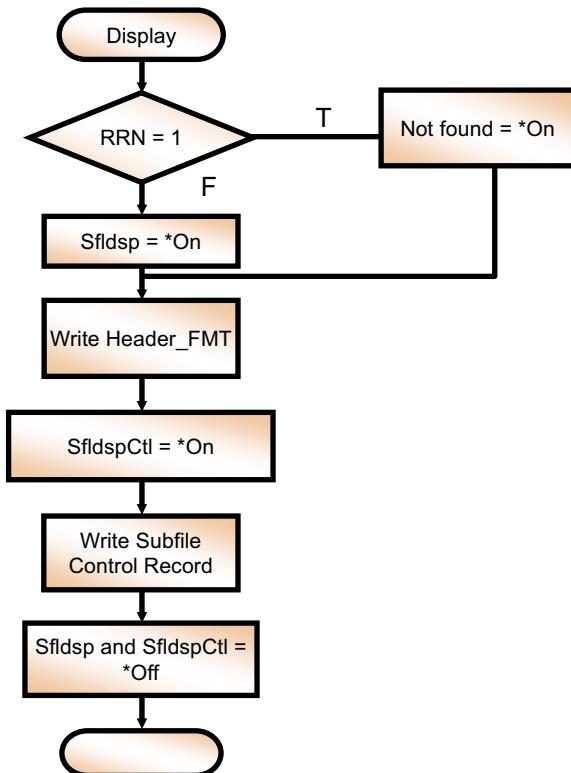
Figure 5-16. Search argument with subfile: Flowchart (2 of 3)

AS075.0

### Notes:

# Search argument with subfile: Flowchart (3 of 3)

IBM i



© Copyright IBM Corporation 2009

Figure 5-17. Search argument with subfile: Flowchart (3 of 3)

AS075.0

## Notes:

# Search argument with subfile: RPG program

IBM i

```

DW 1      Column 1 Replace
....FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords+++++-----+
0001  FItem_PF IF E          K Disk
0002  FItemDspSchCF E        Workstn Sfile(Record:Rrn)
0003  F                           IndDS(WkStnIndices) 000015 /FREE
0004
0005  D WkStnIndices DS
0006  D Exit      3   3N
0007  D SF1End    90  90N
0008  D NotFound  96  96N
0009  D SF1DspCtl 85  85N
0010  D SF1Dsp    95  95N
0011  D SF1Clr    75  75N
0012
0013  D Rrn       S     4   8
000016  Dow not Exit;
000017  SetLL ItmSch Item_PF; // Position file cursor
000018  Read Item_PF;        // using search key
000019  If not %eof(item_PF);
000020    Rrn = 1;
000021  EndIf;
000022
000023  Dow Not %eof(item_PF); // Load subfile with data
000024    Write Record;
000025    Read item_PF;
000026    Rrn = Rrn + 1;
000027  Enddo;
000028
000029
000030  If Rrn = 0;           // Display Subfile - do we have
000031    NotFound = *on;    // any records to display?
000032  Else;
000033    SF1Dsp = *on;      // We have records so set
000034  EndIf;               // SF1Dsp on
000035
000036  Write Heading;
000037
000038
000039  SF1DspCtl = *on;    // Display subfile
000040
000041  SF1DspCtl = *off;
000042  SF1Dsp = *off;
000043  ExFmt Prompt;
000044  NotFound = *off;     // Reinitialize indicator
000045
000046
000047  ExSr SF1Clear;
000048
000049  *InLR = *ON;
000050

000053> 6 Begsr SF1Clear; // Subfile clear subroutine
000054  SF1Clr = *on;
000055  Write Control; // New search - clear subfile
000056  SF1Clr = *off;
000057  Rrn = 0;          // Reset RRN
000058  EndSR;
000059
000060
000061> 1 BegSR *InzSR; // Initialization subroutine
000062  Write FNkeys;
000063  ExFmt Prompt;
000064
000065  SF1End = *on; // Set SF1END indicator
000066  Endsr;
000067 /END-FREE

```

© Copyright IBM Corporation 2009

Figure 5-18. Search argument with subfile: RPG program

AS075.0

## Notes:

We have modified the inquiry program and added a search argument. We also moved the initialization steps into an \*InzSR subroutine. We also add a new subroutine that clears the subfile to prepare for a new search:

1. Initial prompt for the search argument.
2. Position the file cursor based on the search argument. There are three possibilities:
  - If the search argument sets the file cursor to beginning of file, the subfile will contain all database records. Entering no search argument or a key lower than the first valid key will also do this.
  - If the search argument is equal to a valid key, only records from this key to the end of file will be displayed.
  - If the search argument positions the file cursor to end of file, a message is displayed.
3. If we are not at end of file, read a record and increment the RRN. Continue until end of file is reached.

4. If we have no **New** subfile records to display, we set the error message indicator **NotFound (Mapped to \*in96)** on.
5. If we have **New** subfile records to display, we set the **SFLDSP (Mapped to \*in85)** on.
6. After we have displayed the subfile, we prepare for the next search argument by setting the **SFLCLR (Mapped to \*in75)** on and initializing the RRN.

## Other subfile keywords

IBM i

- **SFLINZ:** Initialize subfile
- **SFLLIN:** Manage number of records displayed on a screen
- **SFLDROP:** Truncates data of long record (used with **SFLFOLD**)
- **SFLFOLD:** Displays (folds) long record on multiple lines

© Copyright IBM Corporation 2009

Figure 5-19. Other subfile keywords

AS075.0

### Notes:

These keywords are others that you see in subfile applications, although not as frequently as the ones that we are emphasizing in class:

- **SFLINZ** initializes all of the records in the subfile on an output operation. The fields are initialized to blanks, nulls, zeros, or a constant value if the DFT keyword was used with the field. This is typically used with subfiles for data entry.
- **SFLLIN** can be used to increase the number of records displayed on each page when subfile records are short. **SFLLIN** is associated with a number that indicates the number of spaces between columns of subfile records. The system displays as many complete records as possible on each line, based upon your designated spacing between columns.

When subfile records are short, this keyword can be used to increase the number of records displayed on each page. SFLLIN is associated with a number that indicates the number of spaces between columns of subfile records. The system displays as many complete records as possible on each line, based upon your designated spacing between columns.

**SFLIN** is a record-level keyword used to display subfile records in more than one column on the display. The keyword is associated with a number that indicates the number of spaces between columns of subfile records. Subfile data management displays as many complete records as fit on a line given the designated spacing between those columns.

- **SFLDROP** is a record-level keyword that assigns a CAnn or CFnn key to the function of switching data appearance from a truncated record (enough fields to fit one line) to a folded record (the entire record spanning multiple lines).

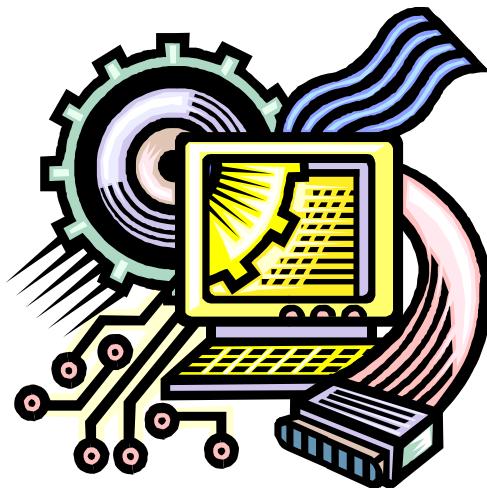
**SFLDROP** assigns a CAnn or CFnn key to act as a toggle switch between displaying records in truncated mode (one per line) and displaying them in folded mode (more than one line per record).

- **SFLFOLD** is a record-level keyword that assigns a CAnn or CFnn key to the function of switching data appearance from a folded record (the entire record spanning multiple lines), to a truncated record (enough fields to fit one line). **SFLFOLD** and **SFLDROP** are mutually exclusive.

**SFLFOLD** is the opposite of **SFLDROP**. With **SFLFOLD**, records appear in folded form first. The user can then switch to truncated form by pressing the assigned CAnn or CFnn key.

# Machine exercise: Inquiry subfiles with search

IBM i



© Copyright IBM Corporation 2009

Figure 5-20. Machine exercise: Inquiry subfiles with search

AS075.0

## Notes:

Perform the machine exercise “Inquiry subfiles with search.”



## 5.3. Subfile maintenance

# Considerations for subfile maintenance

Performance can be critical:

- Size of database file
- Size of subfile
  - Time required to load subfile
- Record locking
  - How to manage

© Copyright IBM Corporation 2009

Figure 5-21. Considerations for subfile maintenance

AS075.0

## Notes:

You now know how to display a subfile as well as how to scroll up and down when **sflsiz>sflpag**. But, we have been dealing with file sizes of 50 and 100 records. What should you consider when working with a DB file of 100,000 records? Should the user be made to wait while your program reads and loads 100,000 records in a subfile?

Even though the system automatically extends the subfile size when **sflsiz>sflpag**, there are some alternatives that we should consider when we design and code a program that will maintain DB records that are displayed in a subfile.

- How should you manage record locking?
- If many users are accessing and maintaining the DB using a subfile, how can the program ensure that the user is seeing the most current copy of the subfile?

# Options

IBM i

- Page at a time processing
- Call programs to perform maintenance outside subfile program
- But...
  - **SFLSIZ = SFLPAG**
  - Scrolling managed in program
  - SFLSIZ not automatically increased
- However...
  - Best possible performance
  - DDS keywords are available
  - Good way to implement record maintenance using subfiles

© Copyright IBM Corporation 2009

Figure 5-22. Options

AS075.0

## Notes:

To optimize performance, we cannot make the user wait while the program loads the complete subfile. *We should present the user only with what the user needs one page at a time.* And, we must write our code in such a way that record locking is minimized and that the latest available subfile data is presented to the user.

What these goals mean is that we write our programs such that maintenance is performed by programs that we will call. And, we manage subfile pages in our program by using the **SFLSIZ=SFLPAG** technique.

We pay a price to do this. We will have to write more complex code that manages scrolling up and down in the program. And, with **SFLSIZ=SFLPAG**, the subfile is not automatically increased in size by the system. We must also manage each and every page. However, the users do not have to endure long delays while the subfile is loaded. And, when performing maintenance, the worst case is that the current page of records may be slightly out of sync until a refresh is performed.

However, the techniques that we are about to show you are an excellent way to implement subfile maintenance. In addition, performance is optimized.

# Step-by-step process to adding maintenance function

IBM i

- Rework existing example to use subroutines for various functions
  - Easy to add new functions as new subroutines
  - Easier maintenance of code
- Add enhancements one at a time:
  - Page+1 and Pagedown
  - Pageup
  - SFLPAG = SFLSIZ
  - Maintenance

© Copyright IBM Corporation 2009

Figure 5-23. Step-by-step process to adding maintenance function

AS075.0

## Notes:

In the previous topic, we used the inquiry program against the **ITEM\_PF** file to discuss how to write an inquiry application. We continue to use the same program. However, we need to first rework the program so that all the steps that we need to process the subfile are performed in individual subroutines.

By approaching our work in this way, the application is easier to understand and therefore, to maintain in the future. For our purposes, it is simple to add new function by adding various subroutines one at a time.

In the rest of this topic, we add functions one at a time, looking at both the DDS changes and the program changes. At the end of each step, you perform a lab task that requires you to make the same change to your lab program that worked with the **VNDNAM\_LF** subfile.

# Modular version of vendor search: DDS

IBM i

```
-----#AAN01N02N03..Name+++++RLen++TDpBLinPosFunctions+++++-----
I00010 A REF(ITEM_PF)
I00020 A C003(03)
I00030 A INDARA
I00040 A OVERLAY
I00050 A R PROMPT
I00060 A 2 7DATE EDTCDE(Y)
I00070 A 2 32'Item List Search'
I00080 A DSPATR(HI)
I00090 A 2 65USER
I00100 A 3 2'Enter item number:'
I00110 A 3 31REFFLD(ITMNBR)
I00120 A 4 7'Press Enter to continue or to go -
I00130 A to top of file'
I00140 A COLOR(BLU)
I00150 A OVERLAY
I00160 A 7 7'Item No'
I00170 A 7 20'Description'
I00180 A 7 50'Qty on Hand'
I00190 A 7 63'Qty on Order'
I00200 A ** Subfile Record Format
I00210 A R RECORD SFL
I00220 A ITMNBR R 0 9 7
I00230 A ITMDESCR R 0 9 20
I00240 A ITMQTYOH R 0 9 52EDTCDE(1)
I00250 A ITMQTYOO R 0 9 66EDTCDE(1)
I00260 A ** Subfile Control format
I00270 A R CONTROL SFLCTL(RECORD)
I00280 A SFLDSPCTL
I00290 A SFLDSP
I00300 A SFLCLR
I00310 A SFLSIZ(0016)
I00320 A SFLPAG(0012)
I00330 A SFLEND(*MORE)
I00340 A OVERLAY
I00350 A ** Function key narrative
I00360 A R FNKEYS 23 7'F3=Exit'
I00370 A ** Message for empty subfile
I00380 >1 A R MSG
I00390 A OVERLAY
I00400 A 12 32'No Records Found'
I00410 A DSPATR(HI)
```

© Copyright IBM Corporation 2009

Figure 5-24. Modular version of vendor search: DDS

AS075.0

## Notes:

First, let's rework the DDS a little. Notice that the ERRMSG keyword has been removed. Instead, we have added the MSG record format. We did this because later, we change the message in the DDS from a constant to a variable when we want more messages to be displayed using this subfile format.

# Modular version of vendor search: RPG IV (1 of 3)

```

000100  FItem_PF  IF  E      K Disk
000200  FItemSubs1 CF  E      Workstn Sfile(Record:Rrn)
000300  F                      IndDS(WkStnIndices)
000400
000500  D WkStnIndices      DS
000600  D  Exit             3   3N
000700  D  Sf1End           90  90N
000800  D  Sf1DspCtl        85  85N
000900  D  Sf1Dsp           95  95N
001000  D  Sf1Clr            75  75N
001100
001200  D  Rrn               S   4  0 INZ
001300>> 1) EmptySfl       S   N
001400
001500
001600  /FREE
001700  DoW not Exit;
001800  ExSR Search;
001900  EndDo;
002000
002100  *InLr = *on;
002200

```

© Copyright IBM Corporation 2009

Figure 5-25. Modular version of vendor search: RPG IV (1 of 3)

AS075.0

## Notes:

Our program has a very simple mainline now. As long as the user does not press F3, we will continue to perform the subroutine **Search**.

Notice that we added an indicator, **EmptySfl**. This indicator is set on when there are no subfile records to display and to control the display of the **Msg** record format.

The mainline of the program simply exits to a subroutine that drives the other subroutines. As before, when the user presses F3, the **Exit** indicator is set and the program ends.

## Modular version of vendor search: RPG IV (2 of 3)

```

002200 // Subroutines
002300 BegSR *InzSR; // Initialization subroutine
002400>>1 Write FNKeys;
002500 ExFmt Prompt;
002600 Endsr;
002900
003000
003100>>2 BegSR Search;
003200 ExSr SFLClear; // Clear subfile for new search
003300
003400 SetLL ItmSch Item_PF; // Position file cursor
003500 // using search key
003600 Read Item_PF; // Read first record after cursor
003700
003800 Rrn = 1; // Rrn set to 1 even if we are at EOF
003900 ExSr Fill; // Fill Subfile
004000 ExSr PromptSch; // Prompt for new search
004100 EndSR;
004200
004300>>3 Begsr Fill;
004400
004500 // Load entire subfile
004600 Dow (NOT %EOF(Item_PF)) AND (Rrn <= 9999); // If already at EOF, will not enter loop
004700 Write Record;
004800 Read Item_PF;
004900 Rrn = Rrn + 1;
005000 Enddo;
005100
005200 EmptySfl = (Rrn <= 1); // No records to display?
005300 SflEnd = %EOF(Item_PF); // Have reached EOF of Item_PF
005400
005500 Endsr;
005600

```

© Copyright IBM Corporation 2009

Figure 5-26. Modular version of vendor search: RPG IV (2 of 3)

AS075.0

### Notes:

No logic changes in our program have been made in this part of the code. We perform exactly the same steps as we did before except we do it in individual subroutines:

- \*InzSR:** The initialization subroutine is unchanged. Its purpose is to display the function key narrative and issue the first prompt for a search key to the Item\_PF file.
- Search:** This subroutine drives the application and performs a number of tasks. Because it can be executed at the beginning and for a new search, it first clears the subfile by executing the **SflClear** subroutine. Next, it uses the search argument entered to position the file cursor. Remember that it is possible to position the cursor at end of file as well as anywhere within the file itself. When the cursor is positioned, the subroutine sets the **Rrn** (to position the subfile cursor to its first record); then it executes the subroutine **Fill** to fill the subfile from the cursor forward. Finally, it executes the **PromptSch** subroutine to prompt for another search argument.
- Fill:** Fills a page of subfile records; notice that the first thing we do is to check whether we are at end of file. If the search key positioned the file cursor of Item\_PF at EOF, we do not fill the subfile. We set the **EmptySfl** indicator on. This displays the **Msg** record of

the DSPL. If we are not at EOF of Item\_PF or we have not filled the subfile (using Rrn <= 9999), we loop until either condition is met. When we leave the loop, we also set the SflEnd indicator (*More...* or *End...*) if we have reached EOF of Item\_PF.

## Modular version of vendor search: RPG IV (3 of 3)

```

...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
5600
5700>>4 Begsr PromptSch;
5800
5900     IF NOT EmptySf1;
6000
6100         Sf1DspCtl = *ON; // Display Subfile
6200         Sf1Dsp = *ON;
6300
6400     Else;
6500
6600         Sf1DspCtl = *ON; // No records to view - display message
6700         Sf1Dsp = *OFF;
6800>>    Write Msg;
6900     Endif;
7000     Write Heading;
7100     Write Control;
7200     ExFmt Prompt;
7300 Endsr;
7400
7500>>5 Begsr SFLClear; // Subfile clear subroutine
7600     Sf1Clr = *on;
7700     Sf1Dsp = *OFF;
7800     Sf1DspCtl = *OFF;
7900     Write Control; // New search - clear subfile
8000     Sf1Clr = *off;
8100 EndSR;
8200 /END-FREE

```

© Copyright IBM Corporation 2009

Figure 5-27. Modular version of vendor search: RPG IV (3 of 3)

AS075.0

### Notes:

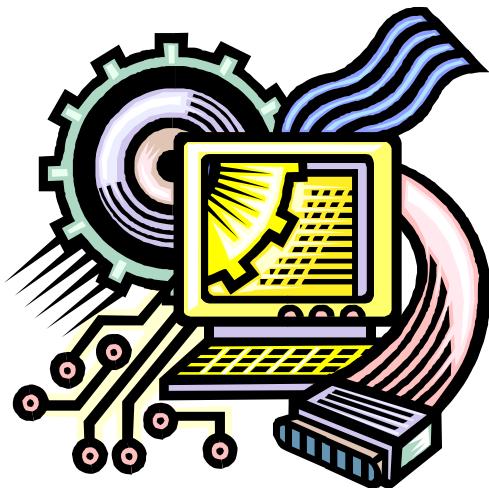
1. **PromptSch:** Notice that, if there are no records to display (perhaps we entered a search argument beyond the end of file), the program now handles it by writing the **Msg** record format, which displays an error message to the user. This subroutine also displays the subfile as loaded by the **Fill** subroutine. It also issues a fresh **Prompt** format that awaits the user to enter a new search argument (or to press F3 to exit the program).

Later, we change the **Prompt** subroutine to prompt the user to allow the user to position the subfile to a specific record for maintenance purposes.

2. **SflClear:** This subroutine clears the subfile. It sets the **Sf1Clr** indicator on the **Sf1Dsp** indicator off (we do not want to display the empty subfile on the display). Notice that we only write the subfile control record to clear the subfile. When done, **Sf1Clr** is set off.

## Machine exercise: Modularize vendor subfile search

IBM



© Copyright IBM Corporation 2009

Figure 5-28. Machine exercise: Modularize vendor subfile search

AS075.0

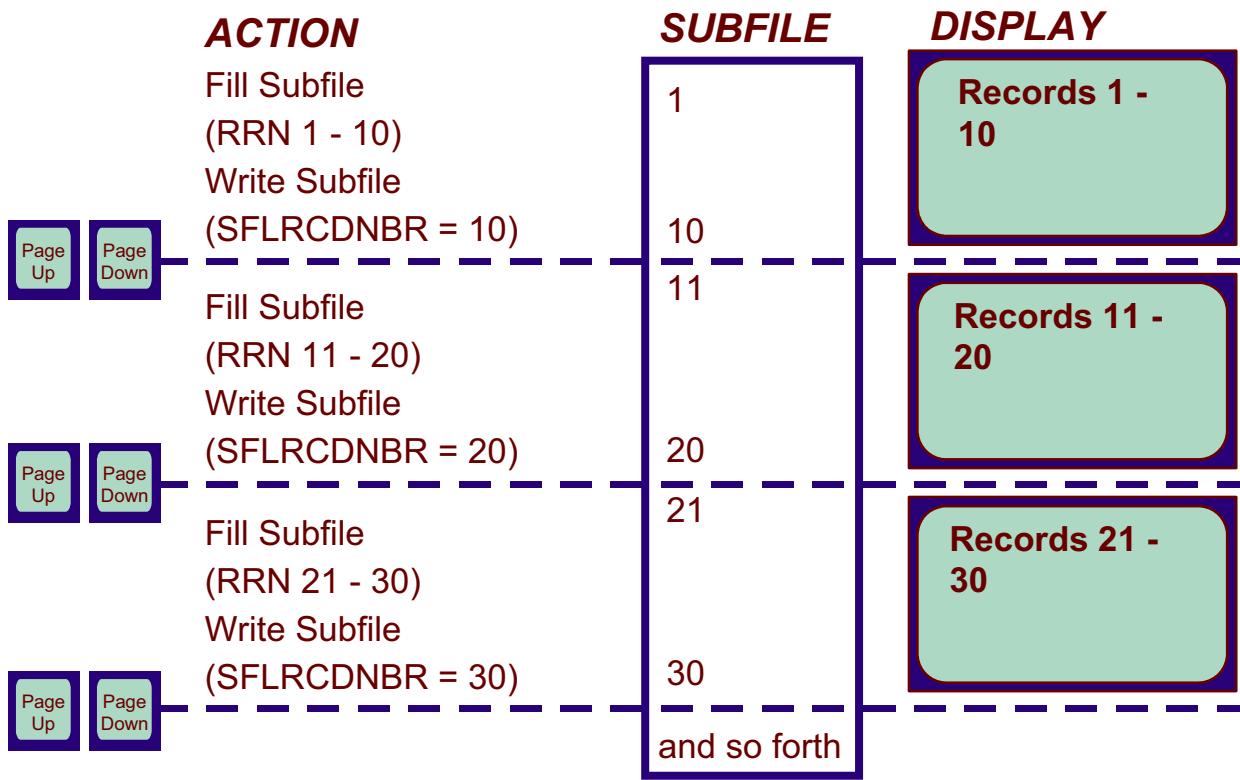
### Notes:

Next, modify your program that you wrote in the previous lab and create a modular solution. Perform the machine exercise “Modularize vendor subfile search.”

## 5.4. Additional maintenance considerations

# Single page subfile load

IBM i



© Copyright IBM Corporation 2009

Figure 5-29. Single page subfile load

AS075.0

## Notes:

Loading all of the records that satisfy the search argument into the subfile at one time forces the user to wait until the subfile has been completely loaded.

Subfile size not equal to page size should be used when a finite number of records can be placed in the subfile and that number is small (for example, 50). The **SFLSIZ** keyword specifies the subfile size. The system allocates space to contain the subfile records based on the value specified for **SFLSIZ**. Specify a value equal to the number of records that you normally have in the subfile. If your program places a record with a relative record number larger than the **SFLSIZ** value into the subfile, *the system extends the subfile to contain it up to a maximum of 9999 records*.

When the *subfile size is not equal to the page size, the use of the Roll Up and Roll Down keys is automatically supported*.

To inform the user that there are more records in the subfile, use the **SFLEND** keyword on the subfile control record.

Alternatively, a program could add records to the subfile one *displayable* page at a time. The user presses the Page Down key. This signals the program that the requested record is not in the current page and the program adds one more page of records to the subfile.

Two keywords, **PAGEDOWN** and **SFLRCDNBR**, are used to implement this technique. The DDS keyword **PAGEDOWN** is the same as **ROLLUP** in function, just as **PAGEUP** is the same as **ROLLDOWN**.

Page-at-a-time processing improves performance. You can also use the i5/OS (OS/400) support to scroll through the subfile. To do this, you define the **ROLLUP** keyword in DDS with a response indicator and also use the **SFLRCDNBR** keyword. In your program, you would write the records needed to fill one subfile page and then display that page. When the user wants to see more records, the user presses the Roll Up key. The program then writes another page of records to the subfile, places the relative record number of a record from the second page into the **SFLRCDNBR** field, and displays the record.

The second page of subfile records is displayed. If the user presses the Roll Down key, the roll down is handled by the system. If the user presses the Roll Up key while the first page is displayed, the system also handles the roll up. The program is notified only when the user attempts to roll up beyond the records currently in the subfile. The program would then handle any additional roll up requests in the same manner as for the second page. When you use this technique, the subfile appears to be more than one page because of the use of the roll keys. Yet, you maintain good response time because the program only fills one subfile page before writing it to the display.



### Note

The **ROLLUP** keyword cannot be specified with **PAGEDOWN**. The **ROLLDOWN** keyword cannot be specified with **PAGEUP**.

**PAGEDOWN** is the same as **ROLLUP**; **PAGEUP** is the same as **ROLLDOWN**.

# Page+1 and Pagedown: DDS (1 of 2)

IBM i

```

ITEMSUB52.DSPF X ITEMSUB52.RPGLE
Line 18 Column 1 Replace
.....AAN01N02N03..Name+++++RLen++TDpBLinPosFunctions+++++
000100 A REF(ITEM_PF)
000200 A CA03(03)
000300 A INDARA
000400 A OVERLAY
000500 A R PROMPT
000600 A 2 7DATE EDTCDE(Y)
000700 A 2 32'Item List Search'
000800 A DSPATR(HI)
000900 A 2 65USER
001000 A 3 2'Enter item number:'
001100 A 3 31REFFLD(ITMNBR)
001200 A 4 7'Press Enter to continue or to go -
001300 A to top of file'
001400 A COLOR(BLU)
001500 A OVERLAY
001600 A 7 7'Item No'
001700 A 7 20'Description'
001800 A 7 50'Qty on Hand'
001900 A 7 63'Qty on Order'
ITMSCH R D I

```

No changes!

© Copyright IBM Corporation 2009

Figure 5-30. Page+1 and Pagedown: DDS (1 of 2)

AS075.0

## Notes:

The PROMPT and HEADING formats are unchanged.

# Page+1 and Pagedown: DDS (2 of 2)

IBM i

```

Line 44      Column 1      Replace
.....AAN01N02N03.....Functions+++++
002000      ** Subfile Record format
002100      A      R RECORD          SFL
002200      A      ITMNBR   R      O 9 7
002300      A      ITMDESCR  R      O 9 20
002400      A      ITMQTYOH R      O 9 52EDTCDE(1)
002500      A      ITMQTYOO R      O 9 66EDTCDE(1)
002600      ** Subfile Control format
002700      A      R CONTROL         SFLCTL(RECORD)
002800 >3 A N90               PAGEDOWN(30)
002900      A 85
003000      A 95
003100      A 75
003200 >1 A
003300      A
003400      A 90
003500      A
003600 >1 A      SFLSIZE    5S 0P
003700 >2 A      SFLRRN     4S 0H      SFLRCDNBR
003800      ** Function key narrative
003900      A      R FNKEYS
004000      A
004100      ** Message for empty subfile
004200      A      R MSG
004300      A
004400      A
004500      A      OVERLAY
12 32 'No Records Found'
DSPATR(HI)

```

© Copyright IBM Corporation 2009

Figure 5-31. Page+1 and Pagedown: DDS (2 of 2)

AS075.0

## Notes:

We make several changes to the control format for our lecture program:

1. Rather than hard coding a value for **SFLSIZ**, we set up a variable **SFLSIZE** which is set to a value that we can modify in the RPG IV program. First, we set it equal to one greater than **SFLPAG**.

**SFLSIZ** dictates the size of each allocation of working storage each time an extension is performed; that is, the subfile fills up. Perhaps the programmer might want to adjust the increment size depending on the amount of data to be loaded:

- Lots of data = lots of paging = more increments => smaller increments
- Less data = less paging = less increments => larger increments

Also, making **SFLSIZ** a variable allows the programmer the flexibility to *drive* the same subfile in different ways, perhaps from different programs where:

- **SFLSIZ = SFLPAG** (no expansion: clear-as-you-page)
- **SFLSIZ > SFLPAG** (expansion: expand-as-you-page)

2. Also, we use a new keyword, **SFLRCDNBR**. This keyword is used to specify that the page of the subfile to be displayed is the page containing the record whose relative record number is in this field. If you do not specify this keyword, i5/OS (OS/400) displays the first page of the subfile by default. This keyword lets the program control exactly what is displayed.

**SFLRCDNBR** is a field-level keyword that is used in the subfile control record format to specify which page of the subfile is to be displayed. The relative record number field that keeps track of records written in the file is associated with this keyword.

This field can be up to four digits in length and must be defined as a zoned decimal whole number. Here, **SFLRRN** is the field and it is specified as hidden, although it can be displayed if you want. The user does not see the hidden field on the display.

Typically, the program adds 1 to the **RRN** field each time a record is written to the subfile. The program stops adding records to the subfile when it has completely filled a page. At this point the value of the field **RRN** equals that of the last record that was written to the subfile. **SFLRRN** is the field that is associated with **SFLRCDNBR**. In this case it is equal to the number of the last record on the page that is being displayed.

3. We specify the **PAGEDOWN** keyword. When page down is pressed, indicator 30 in the DDS is set. This indicator signals the RPG IV program. **PageDown** in the program is mapped to indicator 30. Notice that it is set only when we are not at the end of the subfile (N90). Usually this keyword is specified in the control record format. This means we will have to input the control format, even though it contains no data.

# Page+1 and Pagedown: RPG IV (1 of 3)

```
*ITEMSUBS2.RPGLE X
Line 29      Column 1      Replace 1 change
  . . . . . 1 . . . . 2 . . . . 3 . . . . 4 . . . . 5 . . . . 6 . . . . 7 . . . . 8
000100  FItem_PF  IF   E          K Disk
000200 :>1  FItemSubs2 CF   E          Workstn Sfile(Record:Rrn)
000300      F                           IndDS(WkStnIndices)
000400
000500  D WkStnIndices  DS
000600  D Exit           3     3H
000700 :>2  D PageDown        30    30H
000800  D SflEnd         90    90H
000900  D SflSpCtl       85    85H
001000  D SflDsp          95    95H
001100  D SflClr          75    75H
001200
001300  D Rrn             S          4  0  INZ
001400 :>3  D RrnCount        S          Like(Rrn)
001500 :>1  D EmptySfl        S
001600
001800 /FREE
001900  DoW not Exit;
002000 :>>
002100 :>4  Select;
002200 When PageDown;
002300  Exsr NextPage;
002400 :>>
002500  Other;
002600  Exsr Search;
002700  Endsl;
002800
002900 *InLr = *on;
003000
```

© Copyright IBM Corporation 2009

Figure 5-32. Page+1 and Pagedown: RPG IV (1 of 3)

AS075.0

## Notes:

We again have highlighted changes made to the previous version of the program using **>>**:

1. We define the modified DSPF.
2. The named indicator, **PAGEDOWN**, is mapped to indicator 30, which we specified in the DDS.
3. We define a field, **RRNCOUNT**, that we use to track where we are in the subfile page.
4. We add a select group to process the **PAGEDOWN** option in the mainline. If the user presses **PAGEDOWN**, we handle the request. Otherwise, we execute the **Search** subroutine as the user has entered a new search argument. Remember also that we coded the DDS so that Pagedown would be active only when we have records in the subfile and we are not at EOF of the subfile.

## Page+1 and Pagedown: RPG IV (2 of 3)

```

ITEMSUB52.RPGLE X
Line 62      Column 1      Replace
. . . + . . . 1 . . . + . . . 2 . . . + . . . 3 . . . + . . . 4 . . . + . . . 5 . . . + . . . 6 . . . + . . . 7 . . . + . . . 8

003100      // Subroutines
003200      BegSR *InzSR;      // Initialization subroutine
003300 >>5    Sf1Size = 13;
003400 >>6    Sf1End = *On;
003500      Sf1Spctl = *On;
003600      Write FNKeys;
003700      ExFmt Prompt;
003800      Endsr;
003900
004000      BegSR Search;
004100      ExSr SFLClear;      // Clear subfile for new search
004300      SetLL ItmSch Item_PF; // Position file cursor
004400          // using search key
004500      Read Item_PF;      // Read first record after cursor
004700      Rrn = 1;           // Rrn set to 1 even if we are at EOF
004800      Exsr Fill;         // Fill Subfile
004900      Exsr PromptSch;    // Prompt for new search
005000      EndSR;
005100
005200      Begsr Fill;
005400          // Load next block of records
005500 >>7    RrnCount = 1;
005600          Dow (NOT %EOF(Item_PF))
005700 >>7    AND (RrnCount <= (Sf1Size-1));
005800          Write Record;
005900          Read Item_PF;
006000          Rrn = Rrn + 1;
006100 >>7    RrnCount = RrnCount + 1;
006200      Enddo;
006400      EmptySfl = (Rrn <= 1); // No records to display?
006500      Sf1End = %EOF(Item_PF); // Have reached EOF of Item_PF
006600
006700      Endsr;

```

© Copyright IBM Corporation 2009

Figure 5-33. Page+1 and Pagedown: RPG IV (2 of 3)

AS075.0

### Notes:

In this second part of the code, we make several enhancements to support **Page+1** and **PAGEDOWN**:

1. **SFLSIZ** is set to **SFLPAG** plus 1 (13).
2. **SFLEND** is initialized as on, assuming that the search results in records to display.
3. Notice the logic to manage and test **RRNCOUNT**. Whereas **RRN** is used to determine whether we have at least one data record to display and to see whether we have filled the subfile, **RRNCOUNT** is used to determine whether we have **filled a subfile page**. Both are used in the **Fill** subroutine.

# Page+1 and Pagedown: RPG IV (3 of 3)

```

ITEMSUBS2.RPGLE X
Line 96      Column 1      Replace
006900      Begsr PromptSch;
007100      If NOT EmptySfl;
007200
007300      SflDspCtl = *ON; // Display Subfile
007400      SflDsp = *ON;
007500 >8    SflRrn = Rrn - 1;
007600
007700      Else:
007800
007900      SflDspCtl = *ON; // No records to view - display message
008000      SflDsp = *OFF;
008100      Write Msg;
008200      Endif;
008300      Write Heading;
008400      Write Control;
008500      ExFmt Prompt;
008501      Read Control;
008600      Endsr;
008700
008800 >9    Begsr NextPage;
008900      // Load next block of records
009000      Exsr Fill;
009100 >>
009200 >>
009300 >>
009400      Endsr;
009500      Begsr SFLClear; // Subfile clear subroutine
009600      SflClr = *on;
009700      SflDsp = *OFF;
009800      SflDspCtl = *OFF;
009900      Write Control; // New search - clear subfile
010000      SflClr = *off;
010100      EndSR;
p10200      /END-FREE

```

© Copyright IBM Corporation 2009

Figure 5-34. Page+1 and Pagedown: RPG IV (3 of 3)

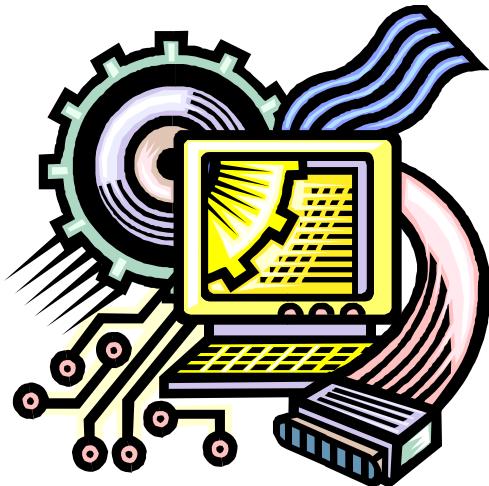
AS075.0

## Notes:

1. **SFLRRN** is the number of the last record on the page that is being displayed, which is always one less than the current value of **Rrn**.
2. The subroutine **Nextpage** has been coded to handle when **PAGEDOWN** is pressed. This routine reads a *page* of data records and fills the subfile for display to the user. Remember, we added an **ExSR** in the mainline to handle **PAGEDOWN** by executing the **NextPage** subroutine. Within **NextPage**, we first execute the **Fill** subroutine, which resets **RrnCount** to **1** and fills the subfile.

# Machine exercise: Page+1 and Pagedown

IBM i



© Copyright IBM Corporation 2009

Figure 5-35. Machine exercise: Page+1 and Pagedown

AS075.0

## Notes:

Perform the machine exercise “Page+1 and Pagedown.”

# Pageup: DDS

IBM i

```

ITEMSUB3.I
Line 28    Column 1    Replace
.....AAN01N02N03.....Functions+++++oooooooooooo
002100    ** Subfile Record format
002200    A      R RECORD           SFL
002300    A      ITMNBR   R      0 9 7
002400    A      ITMDESCR R      0 9 20
002500    A      ITMQTYOH R      0 9 52EDTCDE(1)
002600    A      ITMQTY00 R      0 9 66EDTCDE(1)
002700    ** Subfile Control format
002800    A      R CONTROL          SFLCTL(RECORD)
002900    A      N90                PAGEDOWN(30)
003000 >>  A      N41                PAGEUP(31)  
003100    A      85
003200    A      95
003300    A      75
003400    A
003500    A
003600    A      90
003700    A
003800    A      SFLSIZE     5S 0P
003900    A      SFLRRN     4S 0H      SFLRCDNBR
004000    ** Function key narrative
004100    A      R FNKEYS
004200    A      23 7 'F3=Exit'
004300    ** Message for empty subfile
004400    A      R MSG
004500    A
004600    A
004700    A      OVERLAY
12 32 'No Records Found'
DSPATR(HI)

```

© Copyright IBM Corporation 2009

Figure 5-36. Pageup: DDS

AS075.0

## Notes:

Once again, only the control format has been changed.

The **PAGEUP** keyword is used to tell the program to begin a new subfile search. Again, we define **PAGEUP** within the control format (as we did with **PAGEDOWN**).

Recall that \*IN30 is turned on by the DSPF when the operator presses the **PageDown** key.

DDS sets indicator 31 to tell the program that the user has pressed the PageUp key. Indicator 41 is conditioned by the program. Indicator 41 conditions whether PageUp is active. PageUp is active when indicator 41 is off. This is the opposite situation of Indicator 90 (End of subfile), which conditions PageDown. We use the indicators this way because we do not want to attempt to pageup beyond the beginning of the subfile(41) and we do not want to be able to pagedown below the bottom of the subfile (90).

# Pageup: RPG IV (1 of 4)

IBM i

```
*ITEMSUBS3.RPGLE X
Line 31      Column 1      Replace 2 changes
.....+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
000100      FItem_PF    IF     E          K Disk
000200      >1 FItemSubs3 CF     E          WorkStn Sfile(Record:Rrn)
000300      F          DS
000500      D WkStnIndices   DS
000600      D Exit           3      3N
000700      D PageDown        30     30N
000800      >2 D PageUp          31     31N
000900      D SflEnd          90     90N
001000      >2 D SflBegin        41     41N
001100      D SfldspCtl       85     85N
001200      D Sfldsp          95     95N
001300      D SflClr           75     75N
001400
001500      D Rrn             S          4  0 INZ
001600      D RrnCount        S          Like(Rrn)
001700      D EmptySfl         S          N
001800
002000      /FREE
002100      DoW not Exit;
002200      Select;
002300      When PageDown;
002400      Exsr NextPage;
002500
002600 >>3 When PageUp;
002700 >> Exsr PrevPage;
002800 >> Other;
002900 >> ExSR Search;
003000 >> Ends1;
003100      EndDo;
003200
003300      *InLr = *on;
```

© Copyright IBM Corporation 2009

Figure 5-37. Pageup: RPG IV (1 of 4)

AS075.0

## Notes:

To add **PageUp** in the RPG IV program:

1. We define the modified DSPF, **ItemSubs3**.
2. Map named indicators for **PageUp** (Mapped to \*in31) and **SflBegin** (Mapped to \*in41). Just as we monitored the PageDown key and checked for the end of the subfile, we are notified when the user presses PageUp (Indicator 31) and we need to check when we reach the top of the subfile.
3. In the mainline, we add a **select when** to condition the execution of the **PageUp** subroutine.

## Pageup: RPG IV (2 of 4)

```

ITEMSUBS3.RPGLE X
Line 33    Column 1    Insert
. . . + . . . 1 . . . + . . . 2 . . . + . . . 3 . . . + . . . 4 . . . + . . . 5 . . . + . . . 6 . . . + . . . 7 . . . + . . . 8

003400 // Subroutines
003500 BegSR *InzSR; // Initialization subroutine
003600 Sf1Size = 13;
003700 Sf1End = *On;
003800 Sf1Begin = *On;
003900 Sf1Spctl = *On;
004000 Write FNKeys;
004100 ExFmt Prompt;
004200 Endsr;
004300
004400
004500 BegSR Search;
004600 ExSr SFLClear; // Clear subfile for new search
004800 SetLL ItmSch Item_PF; // Position file cursor
005000 Sf1Begin = *On;
005100 // Read Item_PF; // Read first record after cursor
005200 Rrn = 1; // Rrn set to 1 even if we are at EOF
005300 Exsr Fill; // Fill Subfile
005400 Exsr PromptSch; // Prompt for new search
005500 EndSR;
005600
005700
005800 Begsr CheckBOF; // Check if this is the first record in the file
005900 If %EOF(Item_PF);
006000 Sf1Begin = %EOF(Item_PF);
006100 If %EOF(Item_PF);
006200 SetLL *Start Item_PF;
006300 Endif;
006400 ReadP Item_PF;
006500 Endsr;
006600
006700
006800
006900
007000

```

© Copyright IBM Corporation 2009

Figure 5-38. Pageup: RPG IV (2 of 4)

AS075.0

### Notes:

To implement **PageUp**, we have made quite a few enhancements to the subroutines in the logic of the program:

1. Because we want to use **PageUp**, we must be concerned whether we have reached the top, or beginning of the subfile; so we initialize the **Sf1Begin** indicator (mapped to indicator **41**) to on in the **\*InzSR** subroutine.
2. We added the **CheckBOF** subroutine to see whether we are at the beginning of file of the **Item\_PF**. In the subroutine, we read the current first record in the subfile and, using it as a starting point, reads backward toward the beginning of the DB file. The records from this process are placed into the subfile for display to the user. Even though we use the **%Eof** BIF, it is set to true if we reach **BOF** when we issue the **ReadP** opcode. **ReadP** means to read the file from back to front. **Sf1Begin** is set on when we reach the beginning of the file. Also, in order to position the cursor, we use **\*Start** to position the cursor to the first record if we hit a beginning of file condition. Next, we read the first record.

Because we want to read the DB file backwards, we commented out the original Read in the Search subroutine.

3. We want to know whether we are at the first record in the data file; so we execute the **CheckBOF** subroutine. Notice that the Read has been moved to the CheckBOF subroutine. Because the CheckBOF subroutine reads the previous record to determine if we are at the beginning of the file, we let it read the next record to position the file back at the record that was being processed.

# Pageup: RPG IV (3 of 4)

IBM i

```

ITEMSUB53.RPGLE X
Line 64    Column 1    Replace
. ....+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
007100    Begsr Fill;
007300    // Load next block of records
007400    RrnCount = 1;
007500    Dow (NOT %EOF(Item_PF))
007600        AND (RrnCount <= (SflSize-1));
007700        Write Record;
007800        Read Item_PF;
007900        Rrn = Rrn + 1;
008000        RrnCount = RrnCount + 1;
008100    Enddo;
008300    EmptySfl = (Rrn <= 1); // No records to display?
008400    SflEnd = %EOF(Item_PF); // Have reached EOF of Item_PF
008600    Endsr;
008700
008800    Begsr PromptSch;
009000    If NOT EmptySfl;
009200        SflDspCtl = *ON;      // Display Subfile
009300        SflDsp = *ON;
009400        SflRrn = Rrn - 1;
009500
009600    Else;
009800        SflDspCtl = *ON;      // No records to view - display message
009900        SflDsp = *OFF;
010000    >>7    SflBegin = *ON;
010100        Write Msg;
010200    Endif;
010300    Write Heading;
010400        Write Control;
010500        ExFmt Prompt;
010501        Read Control;
010600    Endsr;

```

© Copyright IBM Corporation 2009

Figure 5-39. Pageup: RPG IV (3 of 4)

AS075.0

## Notes:

1. In the **PromptSch** subroutine, we set the **SflBegin** indicator (mapped to indicator 41, which disables PageUp in the DSPF) to on when there are no records to display based on the search argument entered.

# Pageup: RPG IV (4 of 4)

```

ITEMSUBS3.RPGLE X
Line 132    Column 1      Insert
011500 >8 Begsr PrevPage;
011600 // Find out where I am now in the DB
011700 >9 Chain 1 Record; // Chain to first record in subfile
011800 >> Exsr SflClear;
011900
012000 >10 Set11 ItmNbr Item_PF; // Load previous block of records
012100 >> ReadP Item_PF;
012200 >> RrnCount = 1;
012400 // Read back through the file one page
012500 >11 DOW (NOT %EOF(Item_PF)) AND
012600 >> (RrnCount <= (SflSize-1));
012700 >> ReadP Item_PF;
012800 >> RrnCount = RrnCount + 1;
012900 >> Enddo;
012901
013100 // Prevent PAGEUP if first record
013200 >12 Exsr CheckBOF;
013300 >> Rrn = 1;
013400 >> Exsr Fill;
013500 >> Exsr PromptSch;
013600 >> Endsr;
013700
013800 Begsr SFLClear; // Subfile clear subroutine
013900 SflClr = *on;
014000 SflDsp = *OFF;
014100 SflDspCtl = *OFF;
014200 Write Control; // New search - clear subfile
014300 SflClr = *off;
014400 >13 SflBegin = *OFF; // Set BOF of subfile
014500 EndSR;
014600 /END-FREE

```

© Copyright IBM Corporation 2009

Figure 5-40. Pageup: RPG IV (4 of 4)

AS075.0

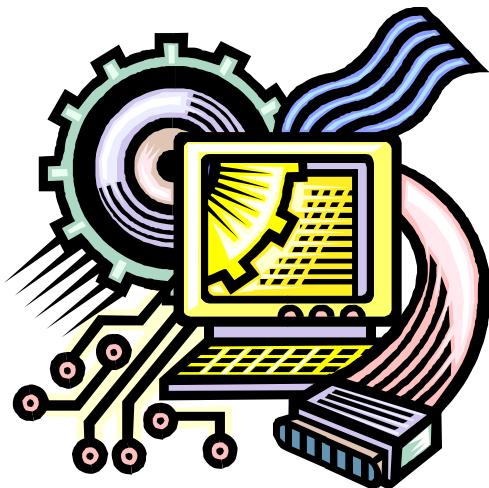
## Notes:

1. We added a subroutine to enable pageup processing. We named it **PrevPage**.
2. In the **PrevPage** subroutine, we need to know where we are in the database file so that we can page up properly. What we do is use a chain to read the first record in the existing subfile. By doing this, we obtain the value of the **ItemNbr**.
3. We use the **ItemNbr** that we retrieved from the subfile to position the cursor in the data file.
4. Now that we have the cursor positioned, we read backwards in the database file for the number of records equal to the size of a subfile page. It is possible that we might reach the beginning of the DB file; so we check for this situation using the **%Eof** BIF. You recall that **RrnCount** keeps track of how records we have read in the DB file. We also include the condition of the Down to check that **RrnCount is <= SflSize -1**. (This is the number of records in SflPage.)

5. We check to see whether we are the beginning of the data file. Then we execute the **Fill** and **PromptSch** subroutines again to fill the subfile and then prompt to write the control record and ask for another search key.
6. In the **SflClear** subroutine, indicators that manage **PageUp** as well as **PageDown** are set off again.

## Machine exercise: Add PageUp

IBM i



© Copyright IBM Corporation 2009

Figure 5-41. Machine exercise: Add PageUp

AS075.0

### Notes:

Perform the machine exercise “Add PageUp.”

## 5.5. Implement SFLPAG = SFLSIZ

# Add SFLPAG = SLFSIZ: DDS

IBM i

```

ITEMSUBS4.DSPF X
Line 19    Column 1    Replace
...+A*.1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
002100    ** Subfile Record format
002200    A      R RECORD           SFL
002300    A      ITMNBR   R      0 9 7
002400    A      ITMDESCR R      0 9 20
002500    A      ITMQTYOH R      0 9 52EDTCDE(1)
002600    A      ITMQTYOO R      0 9 66EDTCDE(1)
002700    ** Subfile Control format
002800    A      R CONTROL          SFLCTL(RECORD)
002900    A      H90               PAGEDOWN(30)
003000 >>  A      H41               PAGEUP(31)
003100    A      85                SFLDSPCTL
003200    A      95                SFLDSP
003300    A      75                SFLCLR
003400    A      SFLSIZE           5S 0P
003500    A      SFLRRN            4S 0H  SFLCDNBR
004000    ** Function key narrative
004100    A      R FNKEYS          23 7 'F3=Exit'
004200    A      R MSG              12 32 'No Records Found'
004300    ** Message for empty subfile
004400    A      R MSG              OVERLAY
004500    A      R MSG              DSPATR(HI)
004600
004700

```

No changes!

© Copyright IBM Corporation 2009

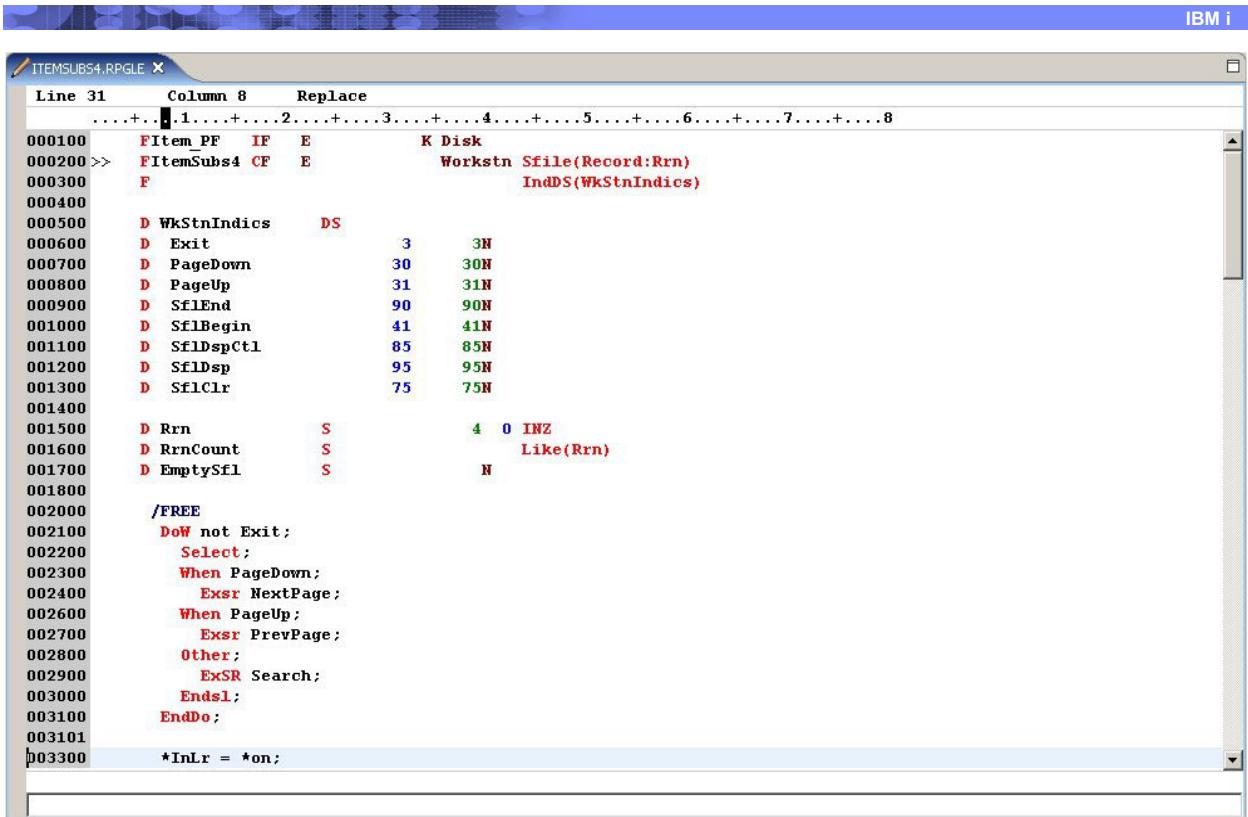
Figure 5-42. Add SFLPAG = SLFSIZ: DDS

AS075.0

## Notes:

No changes to the DDS are necessary.

# Add SFLPAG = SLFSIZ: RPG IV (1 of 5)



```

ITEMSUBS4.RPGLE X
Line 31    Column 8    Replace
.....+...1....+...2....+...3....+...4....+...5....+...6....+...7....+...8
000100    FItem_PF  IF   E          K Disk
000200 >>  FItemSubs4 CF   E          Workstn Sfile(Record:Rrn)
000300        F                               IndDS(WkStnIndices)
000400
000500    D WkStnIndices     DS
000600    D Exit           3      3N
000700    D PageDown        30     30N
000800    D PageUp          31     31N
000900    D SflEnd          90     90N
001000    D SflBegin         41     41N
001100    D SflDspCtl       85     85N
001200    D SflDsp          95     95N
001300    D SflClr           75     75N
001400
001500    D Rrn             S          4  0  INZ
001600    D RrnCount        S          Like(Rrn)
001700    D EmptySfl         S          N
001800
002000 /FREE
002100   DoW not Exit;
002200   Select;
002300   When PageDown;
002400     Exsr NextPage;
002600   When PageUp;
002700     Exsr PrevPage;
002800   Other;
002900     ExSR Search;
003000   EndSL;
003100 EndDo;
003101
003300 *InLr = *on;

```

© Copyright IBM Corporation 2009

Figure 5-43. Add SFLPAG = SLFSIZ: RPG IV (1 of 5)

AS075.0

## Notes:

Even though we did not change the DDS, we did make a copy and called it **ItemSubs4**. We reference **ItemSubs4** in the program.

## Add SFLPAG = SLFSIZ: RPG IV (2 of 5)

```

ITEMSUBS4.RPGLE X
Line 35    Column 1      Insert
. . . + . . . + . . . 2 . . . + . . . 3 . . . + . . . 4 . . . + . . . 5 . . . + . . . 6 . . . + . . . 7 . . . + . . . 8
003500 // Subroutines
003600 BegSR *InzSR; // Initialization subroutine
003700 >>1 Sf1Rrn = 1;
003800 >>2 Sf1Size = 12; // Set Sf1Siz = Sf1Pag = 12 records
003900
004000
004100
004200
004300
004400
004500
004600
004700
004900
005100
005200
005300
005400
005500
005600
005700
005800
005900
006100
006200
006300
006500
006600
006700
006900
007000

BegSR Search;
ExSr SFLClear; // Clear subfile for new search
SetLL ItmSch Item_PF; // Position file cursor
Exsr CheckBOF;
// Read Item_PF; // Read first record after cursor

Rrn = 1; // Rrn set to 1 even if we are at EOF
Exsr Fill; // Fill Subfile
Exsr PromptSch; // Prompt for new search
EndSR;

Begsr CheckBOF;
// Check if this is the first record in the file
ReadP Item_PF;
Sf1Begin = %EOF(Item_PF);
If %EOF(Item_PF);
  SetLL *Start Item_PF;
Endif;
Read Item_PF;
Endsr;

```

© Copyright IBM Corporation 2009

Figure 5-44. Add SFLPAG = SLFSIZ: RPG IV (2 of 5)

AS075.0

### Notes:

1. In the **\*InzSR** subroutine, we set the **SFLSIZE** equal to the value of **SFPAG** in the DDS.
2. We now must manage the value of **Rrn** carefully so we initialize it to 1. Remember that **SFLPAG=SFLSIZ** means that we *must manage all scrolling in the program*. And, fortunately, we have already written most of the code to do this.

## Add SFLPAG = SLFSIZ: RPG IV (3 of 5)

```

ITEMSUBS4.RPGLE X
Line 69    Column 1    Insert
007200
007400
007500
007600
007700 >3 Begsr Fill;
007800 // Load next block of records
007900 RrnCount = 1;
008000
008100
008200
008300
008400 EmptySfl = (Rrn <= 1); // No records to display?
008500 SflEnd = %EOF(Item_PF); // Have reached EOF of Item_PF
008700 Endsr;
008800
008900 Begsr PromptSch;
009100 If NOT EmptySfl:
009300   SflDspCtl = *ON; // Display Subfile
009400   SflDsp = *ON;
009500   SflRrn = Rrn - 1;
009600
009700 Else:
009900   SflDspCtl = *ON; // No records to view - display message
010000   SflDsp = *OFF;
010100   SflBegin = *ON;
010200   Write Msg;
010300 Endif;
010400 Write Heading;
010500   Write Control;
010600   Exfmt Prompt;
010601   Read Control;
010700 Endsr;

```

© Copyright IBM Corporation 2009

Figure 5-45. Add SFLPAG = SLFSIZ: RPG IV (3 of 5)

AS075.0

### Notes:

- Now that **SFLSIZ (=SFLPAG)** is equal to 12 rather than 13, we change the condition to compare **RrnCount** to **SflSize** rather than **SflSize-1**.

## Add SFLPAG = SLFSIZ: RPG IV (4 of 5)

```

ITEMSUB54.RPGLE X
Line 100  Column 1  Insert
      .+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
010900 Begsr NextPage;
011000
011100 // Load next block of records
011200 >>4 Exsr SflClear;
011300 >>4 Rrn = 1;
011400 Exsr Fill;
011500 Exsr PromptSch;
011600 Endsr;
011700
011800 Begsr PrevPage;
011900 // Find out where I am now in the DB
012000 Chain 1 Record; // Chain to first record in subfile
012100 Exsr SflClear;
012200
012300 SetLL ItemNbr Item_PF; // Load previous block of records
012400 ReadP Item_PF;
012500 RrnCount = 1;
012600
012700 // Read back through the file one page
012800 DOW (NOT %EOF(Item_PF)) AND
012900 >>5 (RrnCount <= (SflSize));
013000 ReadP Item_PF;
013100 RrnCount = RrnCount + 1;
013200 Enddo;
013300
013400 // Prevent PAGEUP if first record
013500 Exsr CheckB0F;
013600 Rrn = 1;
013700 Exsr Fill;
013800 Exsr PromptSch;
013900 Endsr;

```

© Copyright IBM Corporation 2009

Figure 5-46. Add SFLPAG = SLFSIZ: RPG IV (4 of 5)

AS075.0

### Notes:

1. We reset **Rrn** to 1 when we return from executing the **SFLClear** subroutine. We must clear the subfile each time a key is pressed as we are now responsible for reloading the subfile because **SLFSIZ =SFLPAG**.
2. As we did in the Fill subroutine, now that **SLFSIZ (=SFLPAG)**, we change the condition to compare **RrnCount** to **SflSize** rather than **SflSize-1**.

## Add SFLPAG = SLFSIZ: RPG IV (5 of 5)

IBM i

The screenshot shows an IBM i RPGLE editor window titled 'ITEMSUBS4.RPGLE'. The code is being edited at Line 137, Column 1, with the mode set to 'Insert'. The code is as follows:

```
Line 137    Column 1    Insert
..... /..1....+....2....+....3....+....4....+....5....+....6....+....7....+....8....+....9....+....0
014000
014100 Begsr SFLClear; // Subfile clear subroutine
014200     SflClr = *on;
014300     SflDsp = *OFF;
014400     SflDspCtl = *OFF;
014500     Write Control; // New search - clear subfile
014600     SflClr = *off;
014700     SflBegin = *OFF; // Set BOF of subfile
014800 EndSR;
014900 /END-FREE
```

© Copyright IBM Corporation 2009

Figure 5-47. Add SFLPAG = SLFSIZ: RPG IV (5 of 5)

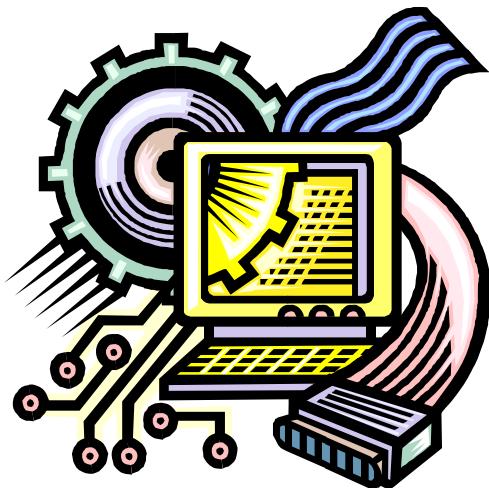
AS075.0

### Notes:

Nothing is changed in the **SFLClear** subroutine.

## Machine exercise: Add SFLPAG = SLFSIZ

IBM i



© Copyright IBM Corporation 2009

Figure 5-48. Machine exercise: Add SFLPAG = SLFSIZ

AS075.0

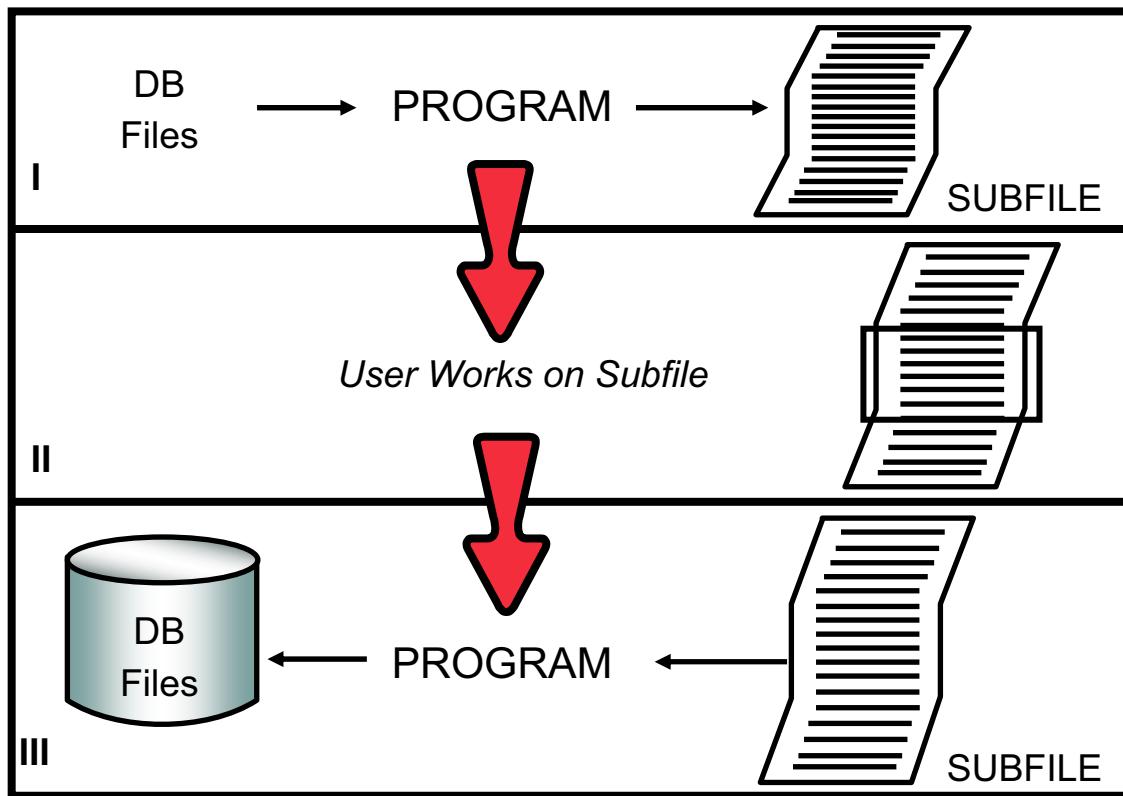
### Notes:

Perform the machine exercise “Add SFLPAG = SLFSIZ.”

## 5.6. Adding file maintenance

# Subfile used for DB maintenance

IBM i



© Copyright IBM Corporation 2009

Figure 5-49. Subfile used for DB maintenance

AS075.0

## Notes:

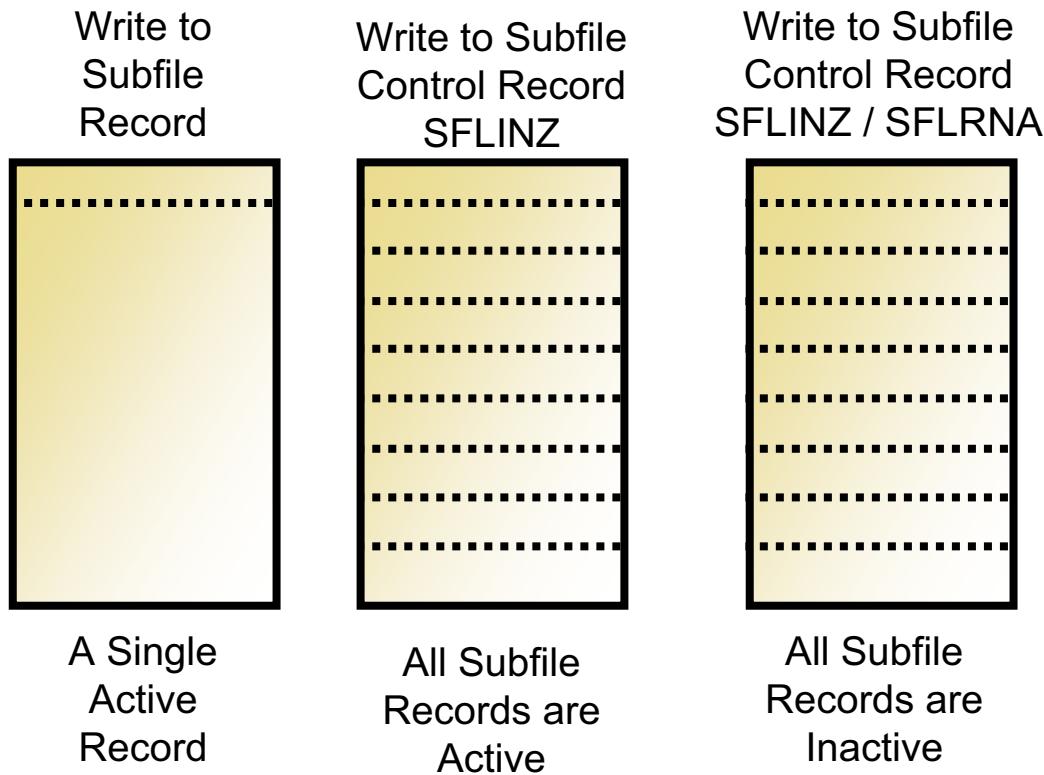
In step I, we perform the same processing as was required for a subfile used for a simple search. The program writes a single record at a time to the subfile. When the program stops writing data records and issues the operation that displays the first page of the subfile, the user can roll through multiple pages, reviewing records that were selected. By placing a selection number in front of the appropriate record, a **CHANGE** or **DELETE** can be selected for any of the viewed records.

When the user presses Enter, step III of subfile processing begins.

If the operator has selected to **CHANGE** or **DELETE** for any records in the subfile, the program must process those transactions and make the appropriate database changes.

# Activation

IBM i



© Copyright IBM Corporation 2009

Figure 5-50. Activation

AS075.0

## Notes:

- **SFLCLR:** Inquiry, or inquiry with update of existing records.

When your program writes a subfile record, that record is considered to be active. If the subfile control record is written with **SFLCLR** activated, the subfile is cleared and no active subfile records exist. The records are not formatted, and you cannot enter new records. This is an approach that can be used for inquiry or inquiry with update. Only records that already exist in the subfile can be updated.

- **SFLINZ:** Data entry.

If your program writes the subfile control record and activates **SFLINZ** all records in the subfile are considered active, and all fields are initialized. Therefore, you can type your new records. This approach is frequently used for data entry.

- **SFLINZ SFLRNA:** Inquiry with update and entry of new records.

The last example shown writes the subfile control record activating **SFLINZ** and **SFLRNA**. All subfile records are not active; so you are allowed to enter new records

and update existing records. This is frequently used when inquiry and data entry of new records are combined.

Inactive subfile records can be keyed into by a user and made active and changed. Inactive records can also be written to with an output operation by your program and made active.

# Maintenance: Design considerations

IBM i

- Must be able to indicate record to be changed
- Where is record?
  - Subfile!
  - Need to get from subfile to DB file
- Must be able to indicate type of change to be made to record:
  - Add new DB record
  - Delete DB record
  - Modify DB record
- Record Locking:
  - Code maintenance inline?
  - Code transactions as individual programs?
- Design maintenance:
  - Modify DSPF to enable subfile record to be changed to be read
  - Modify logic of RPG program (Write and ExFmt opcodes)
  - Keep maintenance code separate; call maintenance programs

© Copyright IBM Corporation 2009

Figure 5-51. Maintenance: Design considerations

AS075.0

## Notes:

To add maintenance to our application, we must make some design changes as well as enhancements.

# Add maintenance: DDS (1 of 2)

```

ITEMSUB5.DSPF X
Line 16    Column 4    Replace
....AAN01N02N03..Name+++++RLen++TDpBLinPosFunctions+++++
000100    A             REF(ITEM_PF)
000200    A             CA03(03)
000300    A             INDARA
000600    A             R PROMPT
000700    A             2 7DATE EDTCDE(Y)
000800    A             2 32'Item List Search'
000900    A             DSPATR(HI)
001000    A             2 65USER
001100    A             3 2'Enter item number:'
001200    A             ITMSCH   R   D   I 3 31REFFLD(ITMNBR)
001300    A             4 7'Press Enter to continue or to go -
001400    A             to top of file'
001500    A             COLOR(BLU)
001600    A             R HEADING
001700    A             ** Heading for Subfile
001800 >1 A             6 2'1=Add 2=Change 4=Delete' DSPATR(HI)
001900 >1 A             7 2'Opt' DSPATR(HI)
002000    A             7 7'Item No'
002100    A             7 20'Description'
002200    A             7 50'Qty on Hand'
002300    A             7 63'Qty on Order'

```

© Copyright IBM Corporation 2009

Figure 5-52. Add maintenance: DDS (1 of 2)

AS075.0

## Notes:

1. We add a heading, **Opt**, to the **Heading** format to be included with the headings for the subfile.

## Add maintenance: DDS (2 of 2)

```

ITEMSUB55.DSPF X
Line 24 Column 1 Replace
....AAN01N02N03..Name+++++RLen++TDpBLinPosFunctions+++++
002400 ** Subfile Record format
002500 A R RECORD          SFL
002600 >-2 A OPTION       1A I 9 3VALUES(' ' '1' '2' '4') DSPATR(HI)
002700 A ITMNR          R 0 9 7
002800 A ITMDESCR        R 0 9 20
002900 A ITMQTYOH        R 0 9 52EDTCDE(1)
003000 A ITMQTY00        R 0 9 66EDTCDE(1)
003100 ** Subfile Control format
003200 A R CONTROL        SFLCTL(RECORD)
003300 A N90              PAGEDOWN(30)
003400 A N41              PAGEUP(31)
003500 A 85               SFLDSPCTL
003600 A 95               SFLDSP
003700 A 75               SFLCLR
003800 A                 SFLSIZE(&SFLSIZE)
003900 A                 SFLPAG(0012)
004000 A 90               SFLEND(*MORE)
004100 A                 OVERLAY
004200 A SFLSIZE          5S 0P
004300 A SFLRRN           4S 0H SFLCDNBR
004400 ** Function key narrative
004500 A R FNKEYS         23 7 'F3=Exit'
004600 A                 12 32 'No Records Found'
004700 ** Message for empty subfile
004800 A R MSG             OVERLAY
004900 A                 12 32
005000 >-3 A*             MESSAGE      25 12 32
005100 >-3 A              MESSAGE      25 12 32 DSPATR(HI)
005200 A

```

© Copyright IBM Corporation 2009

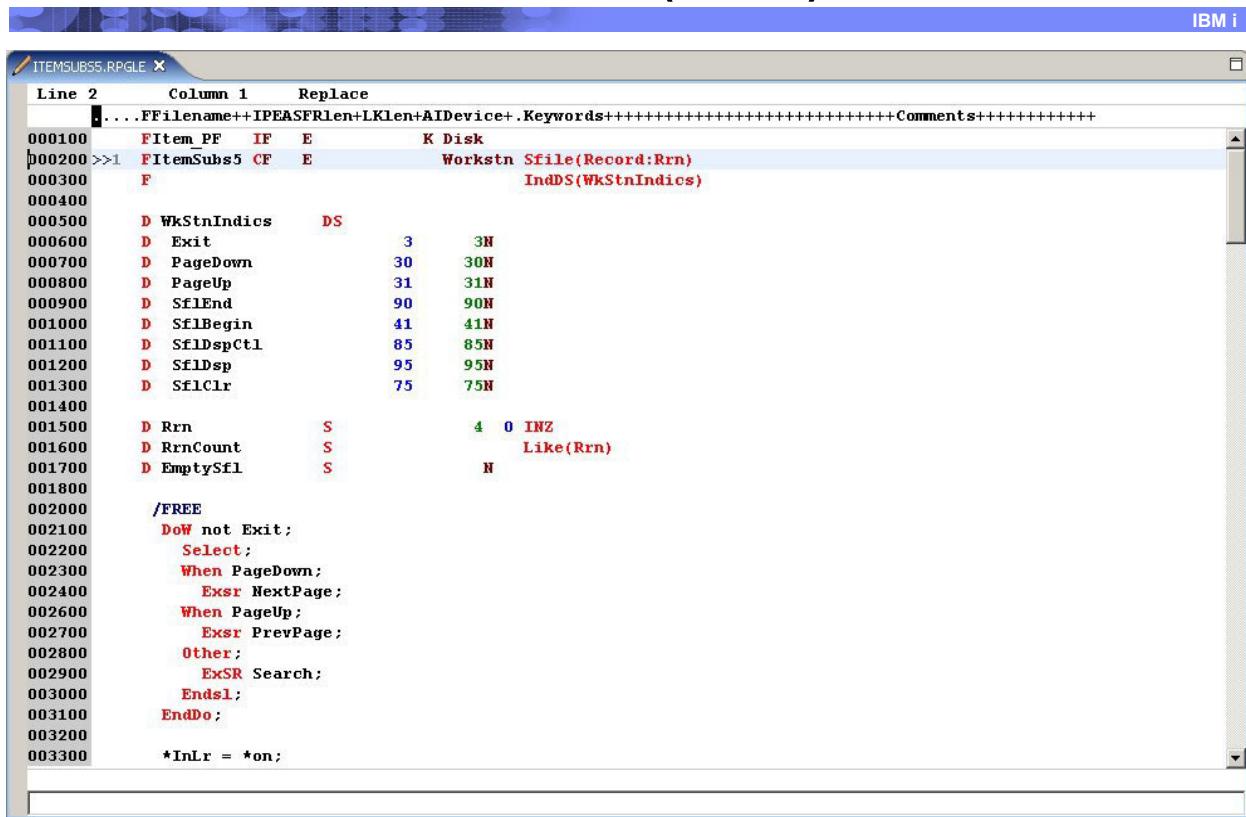
Figure 5-53. Add maintenance: DDS (2 of 2)

AS075.0

### Notes:

2. We add the **Option** field as part of the subfile **Record** format. Note that the only acceptable values are **1 (Add)**, **2 (Change)**, and **4 (Delete)**.
3. For our program, we have selected to call programs to perform maintenance to minimize problems with record locking. We use the **Msg** record format to inform the user that we are performing the desired transaction. We define a new field, **Message**, that holds the value of the message that we want to display.

# Add maintenance: RPG IV (1 of 7)



```

ITEMSUB5.RPGLE X
Line 2      Column 1      Replace
. ....FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords++++++Comments+++++++
000100    FItem_PF   IF   E           K Disk
000200 >>1  FItemSubs5 CF   E           Workstn Sfile(Record:Rrn)
000300    F                           IndDS(WkStnIndices)
000400
000500    D WkStnIndices     DS
000600    D Exit             3      3N
000700    D PageDown         30     30N
000800    D PageUp           31     31N
000900    D Sf1End            90     90N
001000    D Sf1Begin          41     41N
001100    D Sf1DspCtl         85     85N
001200    D Sf1Dsp            95     95N
001300    D Sf1Clr             75     75N
001400
001500    D Rrn               S      4  O INZ
001600    D RrnCount          S      Like(Rrn)
001700    D EmptySf1          S      N
001800
002000 /FREEE
002100   DoW not Exit;
002200   Select;
002300   When PageDown;
002400     Exsr NextPage;
002600   When PageUp;
002700     Exsr PrevPage;
002800   Other;
002900     Exsr Search;
003000   Endsl;
003100   EndDo;
003200
003300 *InLr = *on;

```

© Copyright IBM Corporation 2009

Figure 5-54. Add maintenance: RPG IV (1 of 7)

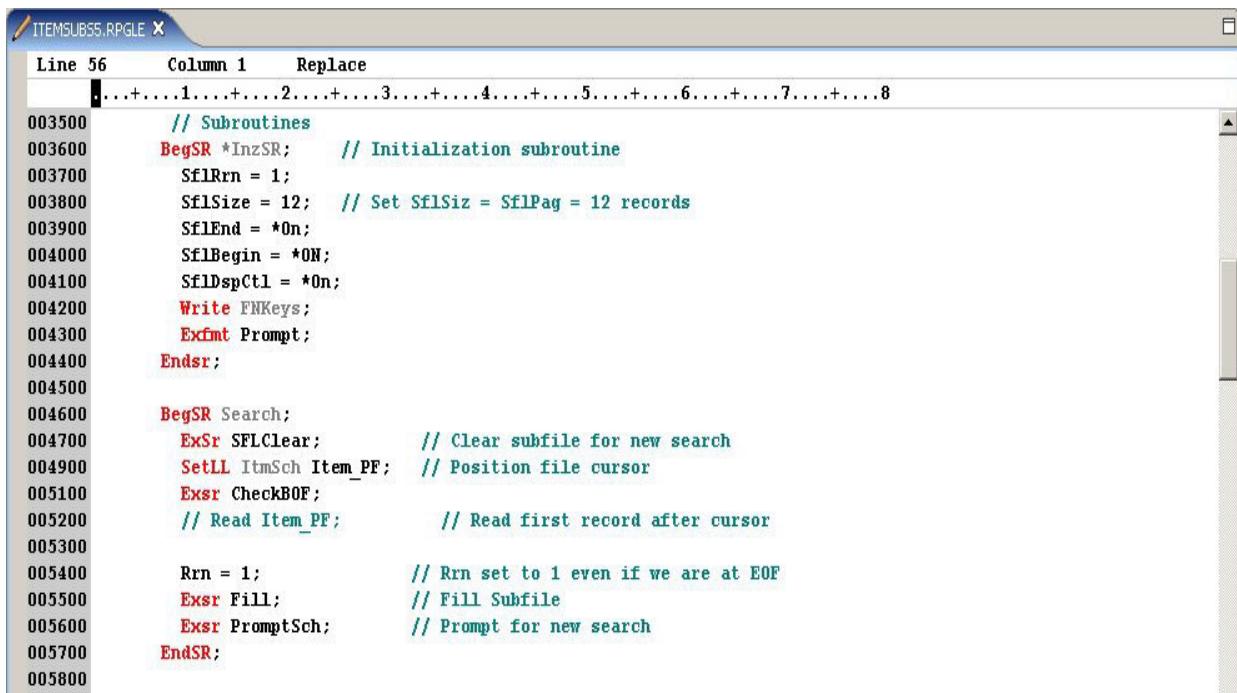
AS075.0

## Notes:

1. Notice that there is no change whatsoever in the mainline code. All we are changing is the DSPF that we use.

## Add maintenance: RPG IV (2 of 7)

IBM i



```

ITEMSUBS5.RPGLE X
Line 56    Column 1    Replace
...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
003500 // Subroutines
003600 BegSR *InzSR; // Initialization subroutine
003700 SflRrn = 1;
003800 SflSize = 12; // Set SflSiz = SflPag = 12 records
003900 SflEnd = *On;
004000 SflBegin = *On;
004100 SflDspCtl = *On;
004200 Write FNKeys;
004300 Exfmt Prompt;
004400 Endsr;
004500
004600 BegSR Search;
004700 ExSr SPLClear; // Clear subfile for new search
004900 SetLL ItmSch Item_PF; // Position file cursor
005100 Exsr CheckBOF;
005200 // Read Item_PF; // Read first record after cursor
005300
005400 Rrn = 1; // Rrn set to 1 even if we are at EOF
005500 Exsr Fill; // Fill Subfile
005600 Exsr PromptSch; // Prompt for new search
005700 EndSR;
005800

```

© Copyright IBM Corporation 2009

Figure 5-55. Add maintenance: RPG IV (2 of 7)

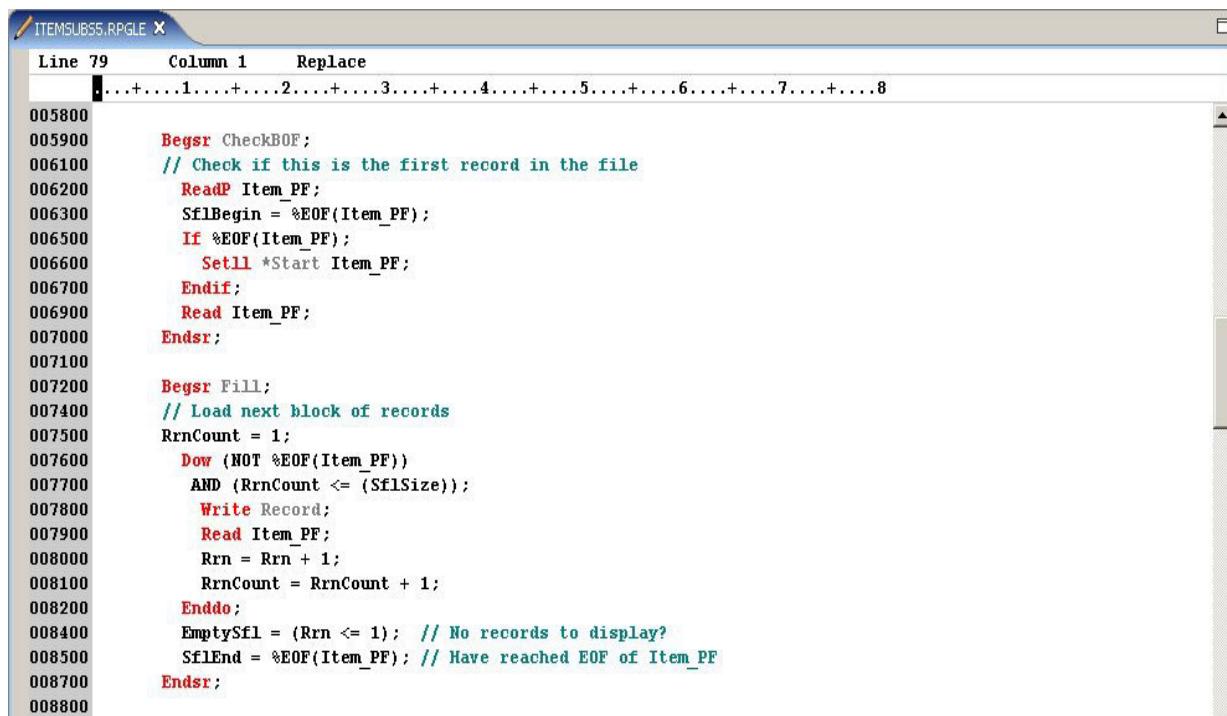
AS075.0

### Notes:

The **\*Inzsr** and **Search** subroutines are unchanged.

## Add maintenance: RPG IV (3 of 7)

IBM i



```

ITEMSUB5.RPGLE X
Line 79      Column 1    Replace
. ....1....2....3....4....5....6....7....8

005800
005900      Begsr CheckBOF;
006100      // Check if this is the first record in the file
006200      ReadP Item_PF;
006300      SflBegin = %EOF(Item_PF);
006500      If %EOF(Item_PF);
006600          Set11 *Start Item_PF;
006700      Endif;
006900      Read Item_PF;
007000      Ends;
007100
007200      Begsr Fill;
007400      // Load next block of records
007500      RrnCount = 1;
007600      Dow (NOT %EOF(Item_PF))
007700          AND (RrnCount <= (SflSize));
007800          Write Record;
007900          Read Item_PF;
008000          Rrn = Rrn + 1;
008100          RrnCount = RrnCount + 1;
008200      Enddo;
008400      EmptySfl = (Rrn <= 1); // No records to display?
008500      SflEnd = %EOF(Item_PF); // Have reached EOF of Item_PF
008700      Ends;
008800

```

© Copyright IBM Corporation 2009

Figure 5-56. Add maintenance: RPG IV (3 of 7)

AS075.0

### Notes:

The **CheckBOF** and **Fill** subroutines are unchanged.

## Add maintenance: RPG IV (4 of 7)

```

ITEMSUBSS.RPGLE X
Line 97    Column 4      Replace
...+....1....2....3....4....5....6....7....8
008900      Begsr PromptSch;
009100      If NOT EmptySfl;
009300      SflDspCtl = *ON; // Display Subfile
009400      SflDsp = *ON;
009500      SflRrn = Rrn - 1;
009700      Else;
009800
009900      SflDspCtl = *ON; // No records to view - display message
010000      SflDsp = *OFF;
010100      SflBegin = *ON;
010200 >>2  Message = 'No records to display';
010300      Write Msg;
010400      Endif;
010500      Write Heading;
010501      Write Control;
010600      ExFmt Prompt;
010700      Read Control;
010800 >>
010900 >>3  If Rrn > 1; // Process changes only if Subfile has records
011000 >>
011100      EndIF;
011200      Endsr;
011300      Begsr NextPage;
011400
011500      // Load next block of records
011600      Exsr SflClear;
011700      Rrn = 1;
011800      Exsr Fill;
011900      Exsr PromptSch;
012000      Endsr;
012100

```

© Copyright IBM Corporation 2009

Figure 5-57. Add maintenance: RPG IV (4 of 7)

AS075.0

### Notes:

2. Because the **Msg** format displays a variable value for Message, we need to assign the value desired. Above, we have an empty subfile with no records to display.
3. In the **PromptSch** subroutine, we also need to process any changes. We code a new subroutine to process the changes. At this point, we **ExSR Changes** to execute the subroutine.

Notice that we check that the subfile has records in it by coding the IF statement to check whether the RRN is greater than 1 (we initialized it to 1 in the **\*InzSR** subroutine).

## Add maintenance: RPG IV (5 of 7)

```

ITEMSUBS5.RPGLE X
Line 140    Column 1      Replace
. ....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
012200      Begsr PrevPage;
012300      // Find out where I am now in the DB
012400      Chain 1 Record; // Chain to first record in subfile
012500      Exsr SflClear;
012600
012700      Set11 ItmNbr Item_PF; // Load previous block of records
012800      ReadP Item_PF;
012900      RrnCount = 1;
013000
013100      // Read back through the file one page
013200      DOW (NOT %EOF(Item_PF)) AND
013300          (RrnCount <= (SflSize));
013400          ReadP Item_PF;
013500          RrnCount = RrnCount + 1;
013600      Enddo;
013700
013800      // Prevent PAGEUP if first record
013900      Exsr CheckBOF;
014000      Rrn = 1;
014100      Exsr Fill;
014200      Exsr PromptSch;
014300      Endsr;
014400
014500      Begsr SFLClear; // Subfile clear subroutine
014600      SflClr = *on;
014700      SflDsp = *OFF;
014800      SflDspCtl = *OFF;
014900      Write Control; // New search - clear subfile
015000      SflClr = *off;
015100      SflBegin = *OFF; // Set BOF of subfile
015200      EndSR;

```

© Copyright IBM Corporation 2009

Figure 5-58. Add maintenance: RPG IV (5 of 7)

AS075.0

### Notes:

The **PrevPage**, **CheckBOF**, and **SflClear** subroutines are unchanged.

## Add maintenance: RPG IV (6 of 7)

```

ITEMSUBS5.RPGLE X
Line 171    Column 1    Replace
015400 >-6 Begsr Changes;
015500 >-7 ReadC Record;
015600
015700 // Process subfile changes
015800 >-8 Dow NOT %EOF(ItemSubs5);
015900 >> Select;
016000
016100 // Add new Item
016200 >-9 When Option = '1';
016300 >> Sf1DspCtl = *ON; // display message
016400 >> Sf1Dsp = *OFF;
016500 >> Sf1Begin = *ON;
016600 >-10 Message = 'Calling Add Program';
016700 >> Write Msg;
016800 >> ExFmt Control;
016900 >> /End-free
017000 >-11 C Call(E) 'ADDPGRAM'
017100 >> /Free
017200 >> If %ERROR;
017300 // Check if Add function completed successfully
017400 >> Endif;
017500
017600 // Change Item details
017700 >-12 When Option = '2';
017800 >> Sf1DspCtl = *ON; // display message
017900 >> Sf1Dsp = *OFF;
018000 >> Sf1Begin = *ON;
018100 >> Message = 'Calling Change Program';
018200 >> Write Msg;
018300 >> ExFmt Control;
018400 >> /End-free

```

© Copyright IBM Corporation 2009

Figure 5-59. Add maintenance: RPG IV (6 of 7)

AS075.0

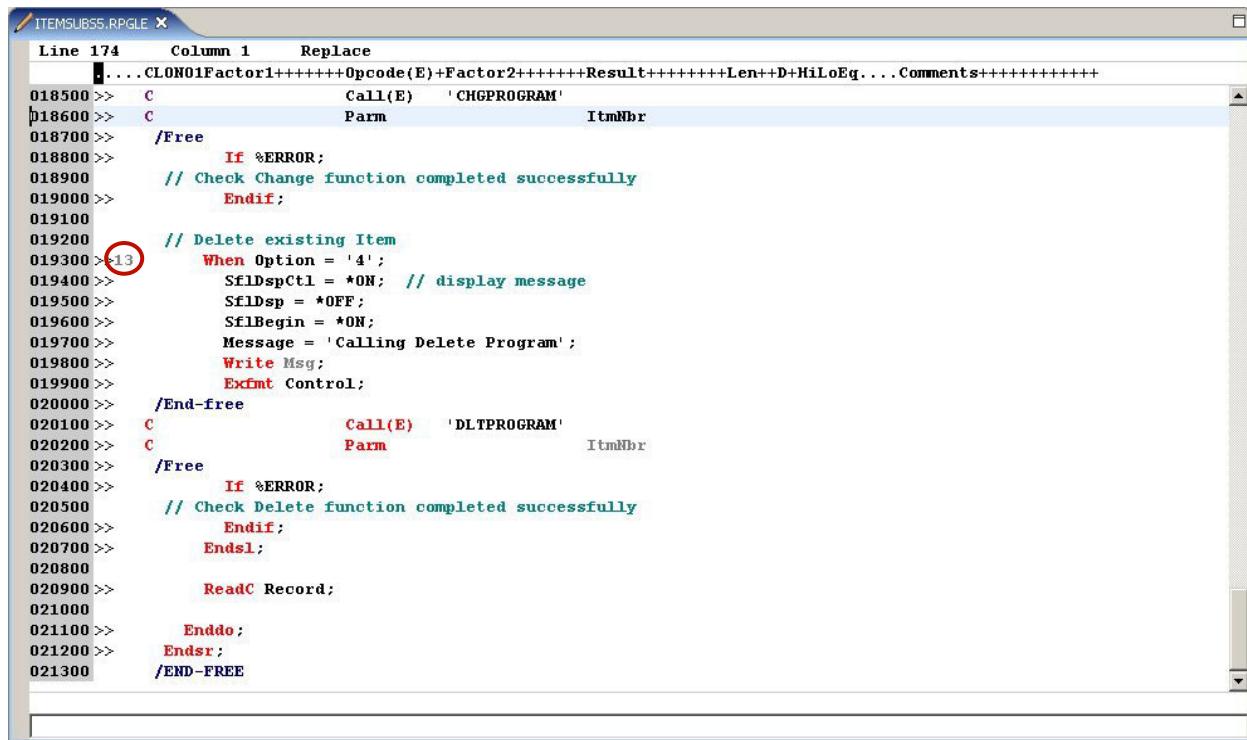
### Notes:

6. The major change to implement maintenance is the **Changes** subroutine. All of the processing could have been handled in line, but we chose to use **CALLs** to individual ADDPROGRAM, CHGPROGRAM, and DLTPROGRAM programs. Performing maintenance in separate programs reduces potential problems with locking and also implements good modular techniques.
7. Notice the use of **ReadC** (read changed). **CHANGES** starts by reading the subfile for changed records.
8. **%Eof** checks whether there are any changes to process. Because we are reading the changes to the subfile, **%Eof** must check the DSPF. Note that if a changed record exists, the select structure handles which type of processing is to follow. Realize that the **READC** has updated the current value of **RRN** with the relative record number of the subfile record that was read.
9. If the user selected option 1 for the subfile record, the Add transaction is processed.

10. For each transaction, we display a message using the appropriate value for Message and followed by a Write of the Msg record format.
11. The ADD program (**ADDPROGRAM**) is called. There are better forms of CALL available to us, which we will discuss later, but for now, this form of CALL is known as a dynamic call. This means that the CALLED program is initiated at the time of the call. Parameters are passed. They are specified by the PARM opcode. We use the E extender and the **%Error** BIF to check whether the call was successful. In our scenario, we did not write the programs that are called. The E extender and %error handle the error and prevent us from experiencing an exception. You learn more about these two in a later unit in this class.
12. If the user selected option 2 for the subfile record, the change program (**CHGPROGRAM**) is called.

# Add maintenance: RPG IV (7 of 7)

IBM i



```

ITEMSUB55.RPGLE X
Line 174    Column 1      Replace
  ....CLON01Factor1+++++0rcode(E)+Factor2+++++Result+++++Len++D+HiLoEq....Comments+++++
018500 >> C           Call(E)      'CHGPROGRAM'
018600 >> C           Parm          ItmNbr
018700 >> /Free
018800 >> If %ERROR;
018900 // Check Change function completed successfully
019000 >> Endif;
019100
019200 // Delete existing Item
019300 >> 13 When Option = '4';
019400 >> Sf1DspCtl = *ON; // display message
019500 >> Sf1Dsp = *OFF;
019600 >> Sf1Begin = *ON;
019700 >> Message = 'Calling Delete Program';
019800 >> Write Msg;
019900 >> ExFmt Control;
020000 >> /End-free
020100 >> C           Call(E)      'DLTPROGRAM'
020200 >> C           Parm          ItmNbr
020300 >> /Free
020400 >> If %ERROR;
020500 // Check Delete function completed successfully
020600 >> Endif;
020700 >> Endsl;
020800
020900 >> ReadC Record;
021000
021100 >> Enddo;
021200 >> Endsr;
021300 /END-FREE

```

© Copyright IBM Corporation 2009

Figure 5-60. Add maintenance: RPG IV (7 of 7)

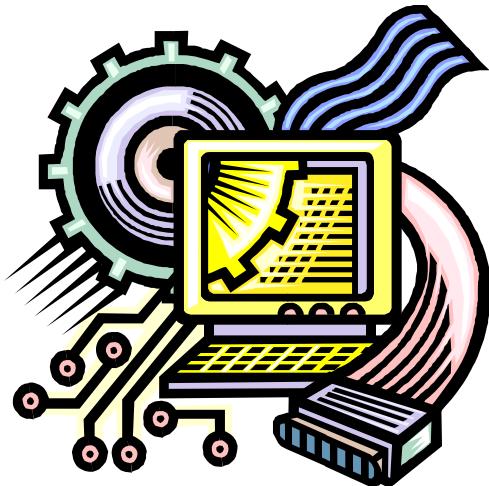
AS075.0

## Notes:

13. If the user selected option 4 for the subfile record, the delete program (**DLTPROGRAM**) is called. Any other value for option is handled as an error in the display file.

## Machine exercise: Add maintenance

IBM i



© Copyright IBM Corporation 2009

Figure 5-61. Machine exercise: Add maintenance

AS075.0

### Notes:

Perform the machine exercise “Add maintenance.”

# Maintenance: Summary

IBM i

- What else can we add to improve the program?
  - Limit option field to one entry per display
  - Use a special function key (other than enter) to submit options
  - Disable entry of search argument when entering option

© Copyright IBM Corporation 2009

Figure 5-62. Maintenance: Summary

AS075.0

## Notes:

## Unit summary

IBM i

Having completed this unit, you should be able to:

- Code inquiry programs to display and search a subfile
- Write RPG IV programs that use SFLSIZ=SFLPAG
- Code page up and page down processing in RPG IV
- Write a subfile maintenance program

© Copyright IBM Corporation 2009

---

Figure 5-63. Unit summary

AS075.0

### **Notes:**

# Unit 6. Managing exceptions and handling errors

## What this unit is about

This unit describes the use of Built-in Functions and Monitor Groups to help you to manage, and sometimes avoid, exceptions in your RPG IV programs.

This unit teaches you how to incorporate basic RPG IV error handling code into programs. It reviews the default error handler and why you should write code that avoids it.

This unit also discusses several approaches to managing program exceptions in an RPG IV program:

- How to use the E opcode extender with an error handling BIF or by setting an error indicator when supported by the opcode to indicate that the error is handled by your RPG IV program
- How to use several error handling Built-in Functions to assist your efforts to determine a solution or workaround to a program problem
- How to use Monitor groups to handle errors

## What you should be able to do

After completing this unit, you should be able to:

- Use the E opcode extender
- Use the ,%open, %found, and %status BIFs as part of your error handling code
- Code Monitor Groups to manage errors

## How you will check your progress

- Machine exercises

## References

SC09-2507	<i>ILE RPG for i5(iSeries) Programmer's Guide, Chapter 12, Handling Exceptions</i>
SC09-2508	<i>ILE RPG for i5(iSeries) Reference, Chapter 5, File and Program Exception/Errors</i>

# Unit objectives



IBM i

After completing this unit, you should be able to:

- Use the E opcode extender
- Use the %error, %open, %found, and %status BIFs as part of your error handling code
- Code monitor groups to manage errors

© Copyright IBM Corporation 2009

---

Figure 6-1. Unit objectives

AS075.0

## Notes:

## 6.1. Using BIFs to avoid errors

## BIFs and errors

IBM i

- %error (and E-extender)
- %status: What is the value of the file status or program status code?
- %open: Is the file (PF, LF, PTRF, DSPF) open?
- %found: Did the program find a record that matches the key?
- %eof: Has the record cursor encountered end of or beginning of the file?
- %equal: Has an exact match been found for the record key?

© Copyright IBM Corporation 2009

Figure 6-2. BIFs and errors

AS075.0

### Notes:

You have already been using several of the BIFs in the visual, such as to determine whether a record was found as a result of a CHAIN. Although not a *pure* error handling BIF, you have been using the **%Found** BIF to decide what to do when you find a record and what to do when you do not find a record. When you are about to add a new record but you find that the key already exists, you have handled a potential duplicate key error.

In AS06/S6197, we used some of the BIFs in the visual. In this topic, we discuss them as they relate to the subject of avoiding errors.

In the next topic, we show you more examples of using BIFs to handle, or, to avoid, errors.

# Summary of I/O BIFs

IBM i

BIF	Parameter	Parameter required?	Result
%EOF	File name	No	'0' or '1'
%EQUAL	File name	No	'0' or '1'
%ERROR	- None -	N/A	'0' or '1'
%FOUND	File name	No	'0' or '1'
%OPEN	File name	Yes	'0' or '1'
%STATUS	File name	No	Unsigned Int (5,0)

© Copyright IBM Corporation 2009

Figure 6-3. Summary of I/O BIFs

AS075.0

## Notes:

Results of I/O operations and certain non I/O operations can be queried by using built-in functions.

If a file name is specified, these functions apply to the last operation on the specified file if the file name is present. Otherwise, they apply to the last relevant operation.



### Note

It is always best to qualify these BIFs with a specific file name. Otherwise, you might not really know which operation on which file set the BIF.

## %Eof

IBM i

```

00001 // File declarations
00002 FPOItem_LF IF E K Disk
00003 FPopLnItmS 0 E Printer OFlind(Overflow)
00004
00005 // Work variable declarations
00006 D LineItem S Like(ItmNbr)
00007 D PoLineCost S +2 Like(PoItmCost)
00008 D SubCost S +3 Like(PoItmCost)
00009 // First-time processing
00010 /FREE
00011   Read POItem_LF;
00012   Write Head_Fmt;
00013   LineItem = ItmNbr;
00014
00015 DoW Not %Eof(POItem_LF);
00016
00017 // Accumulate subtotal values before printing details
00018   PoLineCost = PoQty00 * PoItmCost;
00019   SubCost = SubCost + PoLineCost;
00020
00021   Read POItem_LF;
00022 EndDo;
00023
00024 *InLR = *on;
00025 /END-FREE

```

© Copyright IBM Corporation 2009

Figure 6-4. %Eof

AS075.0

### Notes:

**%Eof** returns a ‘1’ if the most recent READ operation or WRITE operation to a file ended in an end-of-file or beginning of file condition.

This BIF can be used with Read and Write operations, including subfile operations as follows:

- READ (Read Record)
- READC (Read Next Changed Record)
- READE (Read Equal Key)
- READP (Read Prior Record)
- READPE (Read Prior Equal)
- WRITE (Create New Records); subfile only

# %Equal

IBM i

```

000100 // PO Transaction File
000200 FPoTrans_PFIF E Disk
000300 // Open Line Items File
000400 FP00pnLI_LFUF A E K Disk Rename(POLine_FMT : OpenItems)
000500 // Maintenance Printer File
000600 FPOPMaint O E Printer OFLInd(Overflow)
000700 // Build the Line Item Key
000800 D LitmKey DS LikeRec(OpenItems : *key)
000900 /FREE
001000 Read POTrans_PF;
001100 DoW not %eof(POTrans_PF);
001200 // Process transactions
001300 Select;
001400 When TrnCode = 'A';
001500 Exsr AddSr;
001600 When TrnCode = 'C';
001700 Exsr ChgSr;
001800 When TrnCode = 'D';
001900 Exsr DelSr;
002000 Other;
002100 Exsr InvSr;
002200 EndSL;
002300 EndDo;
002400
002500
002600 // Additions
002700 BEGSR ADDSR;
002800 LitmKey.PoNbr = POTPONbr;
002900 LitmKey.ItmNbr = POTitmNbr;
003000 SetLL %Kds(LitmKey) OpenItems;
003100 // If we get an equal condition, error!
003200>> If not %equal(PO0PNLI_LF),
003300 ponbr = potponbr;
003400 ITMNBR = potitmnbr;
003500 Else;
003600 // Handle error for duplicate add here
003700 EndIf;
003800 EndSR;

```

© Copyright IBM Corporation 2009

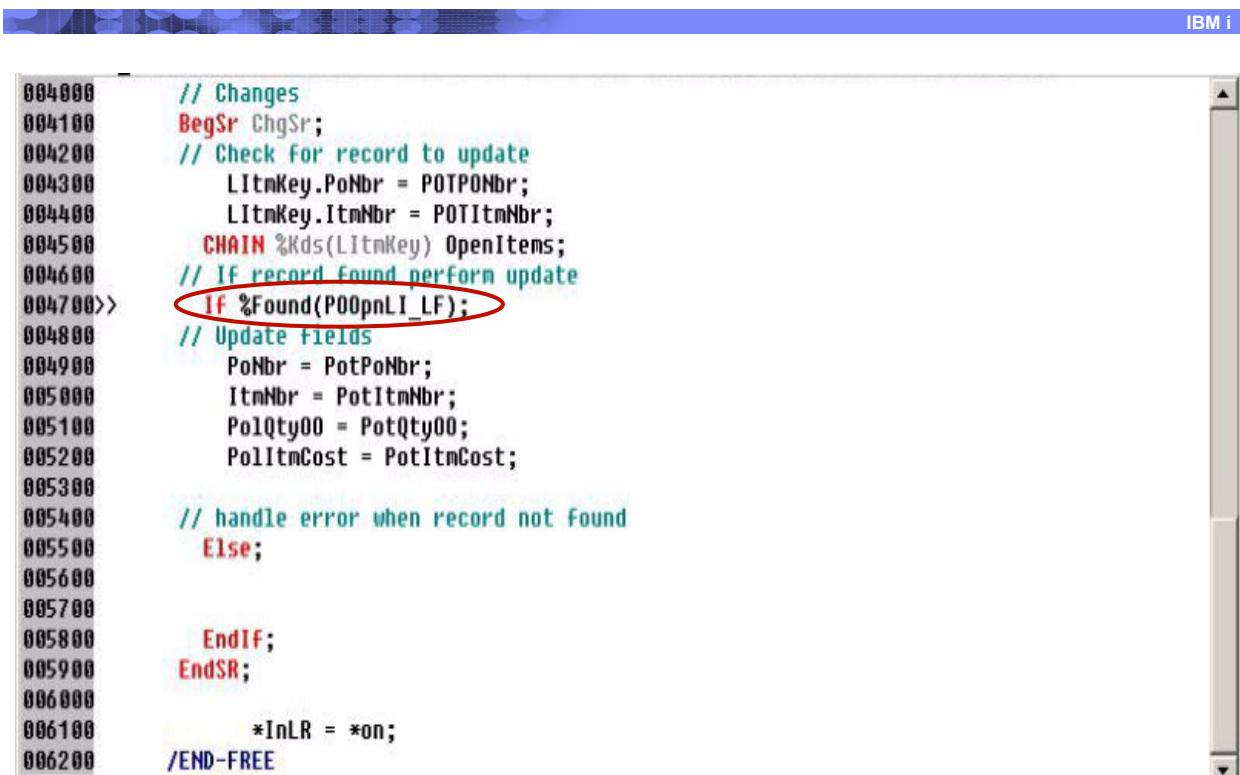
Figure 6-5. %Equal

AS075.0

## Notes:

**%EQUAL** returns a ‘1’ if the most recent operation resulted in an exact match. In the example, if we find a match for the key, *LitmKey.%EQUAL* returns a value of ‘1’. In this program finding a match to the key for an add transaction is an error that we handle in our program.

## %Found



```

004000      // Changes
004100      BegSr ChgSr;
004200      // Check for record to update
004300      LItmKey.PoNbr = POTPoNbr;
004400      LItmKey.ItmNbr = POTItmNbr;
004500      CHAIN %Kds(LItmKey) OpenItems;
004600      // If record found perform update
004700>    IF %Found(P00pnLI_LF);
004800      // Update fields
004900      PoNbr = PotPoNbr;
005000      ItmNbr = PotItmNbr;
005100      PoQty00 = PotQty00;
005200      PoItmCost = PotItmCost;
005300
005400      // handle error when record not found
005500      Else;
005600
005700
005800      EndIF;
005900      EndSR;
006000
006100      *InLR = *on;
006200      /END-FREE

```

© Copyright IBM Corporation 2009

Figure 6-6. %Found

AS075.0

### Notes:

In the example, if a record is found, **%Found** returns a value of '1' when the CHAIN is successful.

The file operations that set **%FOUND** are:

- CHAIN
- DELETE
- SETGT
- SETLL

## %Error and %Found with SETLL

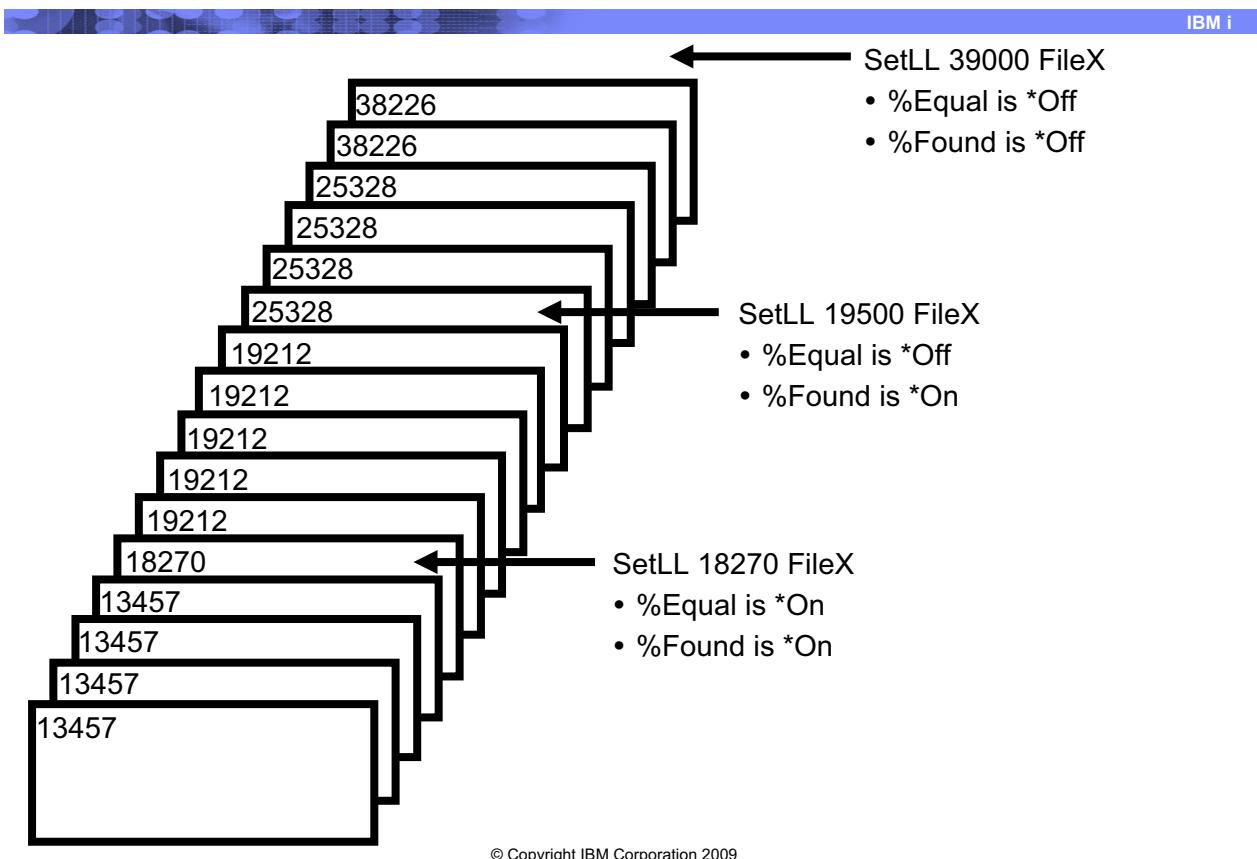


Figure 6-7. %Error and %Found with SETLL

AS075.0

### Notes:

This is a reminder of how the %Equal and %Found BIFs are used in conjunction with the SETLL operation.

# %Open

IBM i

```
00001 /Free
00002>>1 IF not %Open(DBfile);
00003   Open DBfile;
00004 EndIF;
00005
00006
00007>>2 Open(E) DBfile;
00008 If %Error;
00009
0010 /End-Free
```

© Copyright IBM Corporation 2009

Figure 6-8. %Open

AS075.0

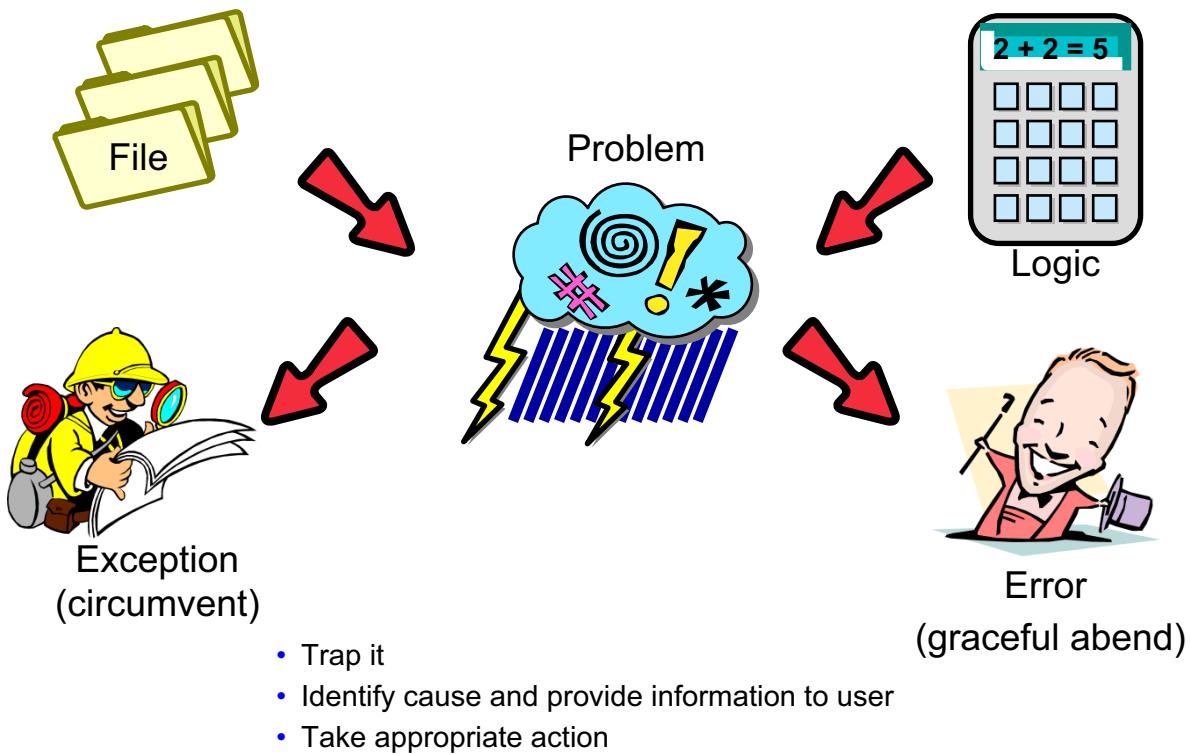
## Notes:

**%Open** returns ‘1’ if the specified file is open.

In the example, it costs much less to do option **1** than **2**, because we only open the file if it is not already open. However, there is a byte in the INFDS that could be checked as easily as **%open**; so **%open** is really just a convenience.

# Program problems

IBM i



© Copyright IBM Corporation 2009

Figure 6-9. Program problems

AS075.0

## Notes:

Coding exception-handling logic in your programs reduces the number of abnormal ends; namely, those associated with function checks.

Error handling can be included to your RPG IV code in anticipation of possible problems or they can be added as workarounds after an application has been placed into production. In general, errors and exceptions can be classified as:

1. File errors, such as file not found, file or record locked, and so on.
2. Program errors, such as division by zero and decimal data errors.
3. Logic errors (errors that we code ourselves). Often these errors are discovered and corrected during testing and debugging. However, sometimes they manage to remain and haunt us during production.

# Exceptions and methods

IBM i

- Some possible exceptions:
  - Duplicate key
  - Data decimal error
  - Divide by zero
  - Called program not found
- Processing halts and message issued:
  - \*ESCAPE: severe
  - \*STATUS: provides program status
  - \*NOTIFY: seeks reply from user
  - Function check: unhandled exception
- Some possible actions:
  - Save totals
  - Identify the record last processed
  - Notify operator
  - Close all open files or data areas
  - Implicit exit to error-handling subroutine

© Copyright IBM Corporation 2009

Figure 6-10. Exceptions and methods

AS075.0

## Notes:

### What do you want to happen when:

- Your program experiences a duplicate key errors?
- You encounter a Decimal Data error?
- You experience an error caused by a division by zero?
- A called program is not found?

Run-time errors cause the system to suspend your program and issue a message to the interactive user (or the system operator if the program is run in batch) of your program.

If you do not address potential data or logic problems in your code, you encounter errors and *processing halts*.

*Exception handling* is the process that involves one or more of these steps:

- *Examine* an exception that has been generated as a result of a run-time error
- *Modify* the exception by handling it

- *Recover* from the exception by passing the exception information to specific code in the program that caused the problem or, perhaps, to another program to take any necessary actions.

When a run-time error occurs, an exception message is generated. An exception message has one of the following types depending on the error which occurred:

**\*ESCAPE:** Indicates that a severe error has been detected.

**\*STATUS:** Describes the status of work being done by a program.

**\*NOTIFY:** Describes a condition requiring corrective action or reply from the calling program.

**Function Check:** Indicates that one of the three previous exceptions occurred and was not handled.

# Methods of managing exceptions

- Handled by programmer:
  1. The E opcode extender and the %Error BIF
  2. Monitor group
  3. ILE condition handler
  4. Implicit exit to error subroutine
- Handled by IBM i:
  5. Default error handler

*Anticipate errors*

*Handle errors in your code*

*Minimize user panic*

© Copyright IBM Corporation 2009

Figure 6-11. Methods of managing exceptions

AS075.0

## Notes:

Unless you write specific error handling code in your RPG IV programs, IBM i invokes the **Default Error Handler** to handle any program exceptions. The **Default Error Handler** can manage the entire exception process. However, the user is presented with an error message that he does not know how to handle and your program more than likely has to be cancelled.

When the **Default Error Handler** assumes responsibility for handling an error, either you have written no error handling code in your RPG IV program or you have exhausted the possibilities handled by your code.

Using the **E-extender and %Error** is a technique that gives you control of the action taken and the messages displayed when a program or file exception is encountered.

However, you can avoid passing control in an exception situation in many cases by coding some basic logic in your programs. A list of potential actions follows:

1. Use the **E** opcode extender and the **%Error** BIF. This technique requires that the opcode supports this method.

2. Code a **monitor group**.
3. Code an **ILE Condition Handler** - this can be a more complex method and should be used in those situations where the first two methods do not address the exception in an adequate manner.
4. Exit to an Error Subroutine, INFSR or \*PSSR. This method is not recommended. The recommended methods are the first two in the list.

# IBM i logic flow for error handling

IBM i

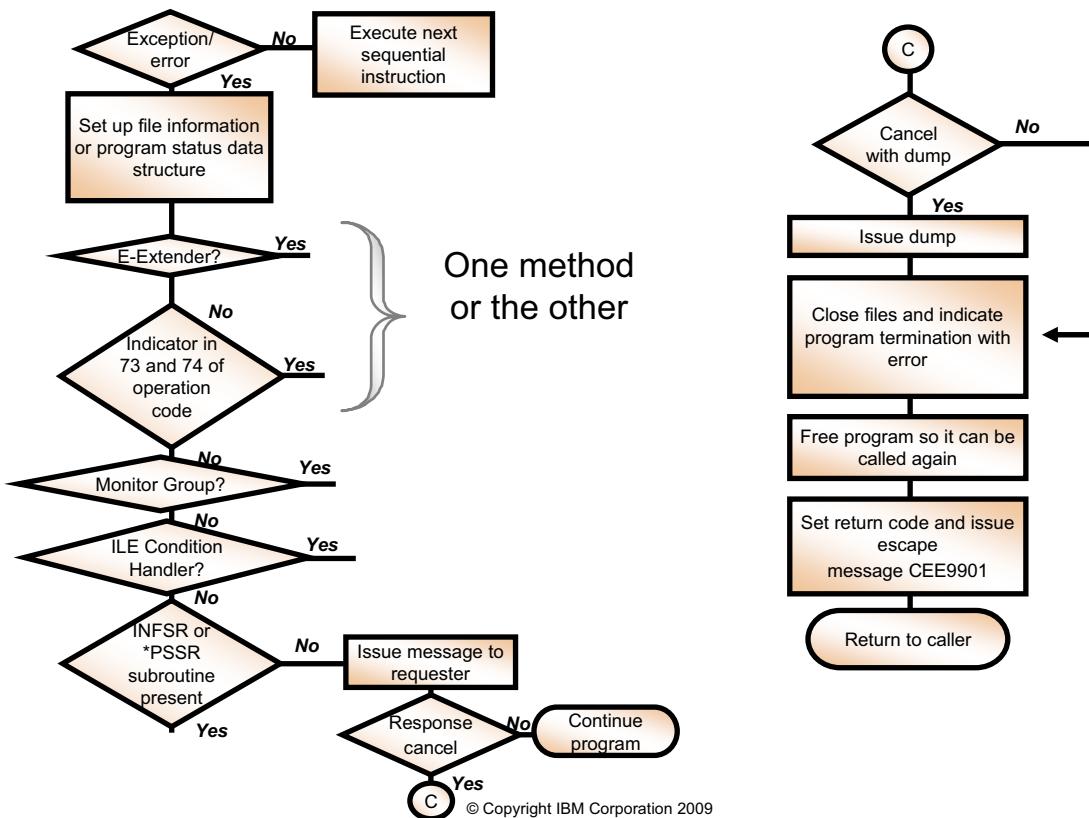


Figure 6-12. IBM i logic flow for error handling

AS075.0

## Notes:

This flowchart illustrates the sequence of events followed by the Default Error Handler. Notice that if no other method is selected, a system-supplied message is sent to the operator who is given the option to cancel the program. If cancel is selected, files and data areas of the active program are closed and control is returned to the calling program. If there was a calling program, it too, would experience a system-supplied message and would halt, giving the user an opportunity to cancel it as well.

Your users see a cryptic system message that does not provide a recommended course of action. It would be better for your application to take control when an exception occurs and override any default error message with a more informative and easily understood message.

The **Yes** branches from the decision blocks mean that your program has told IBM i that your code handles this specific error.

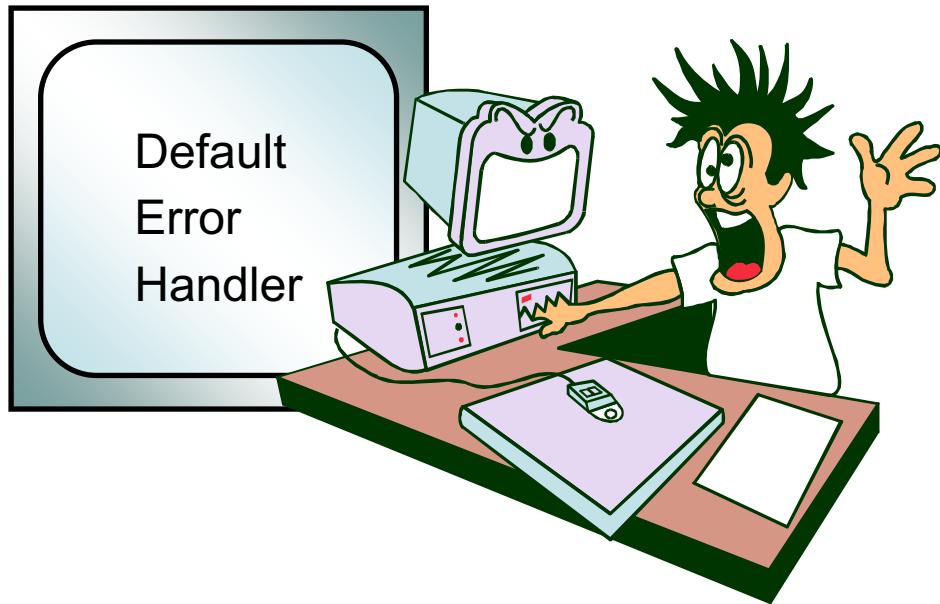
Notice that the flowchart shows the priority of the exception handlers:

1. The E-extender handler and %error (or an error indicator; obsolete).

2. MONITOR group.
3. ILE condition handler.
4. I/O error subroutine handler (for file errors) and Program error subroutine handler (for all other errors); both methods are obsolete.
5. Default handler for unhandled exceptions.

# Default Error Handler

IBM i



© Copyright IBM Corporation 2009

Figure 6-13. Default Error Handler

AS075.0

## Notes:

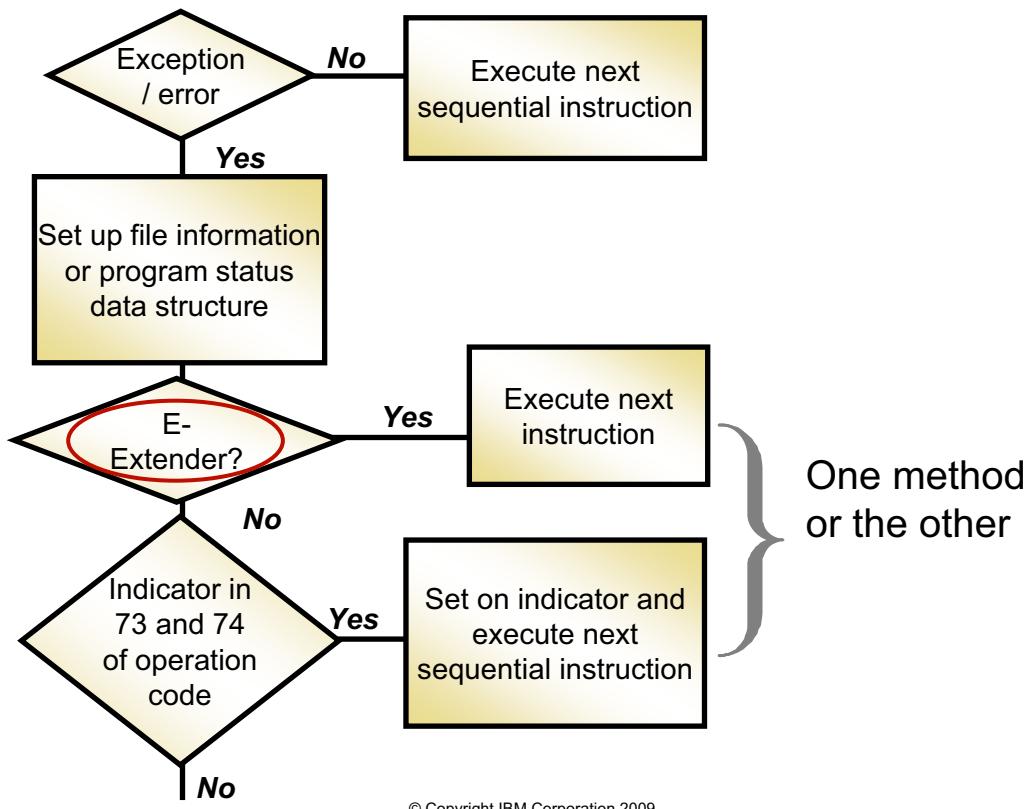
It is good that the Default Error Handler exists.

However, when you consider how intimidating some of the system messages can be to an end user, you realize that you should try to manage the error or exception where possible in your application logic.

The methods of managing exceptions that we discuss in this topic look at alternatives to give your programs more control over what happens.

# Using E-Extender

IBM i



© Copyright IBM Corporation 2009

Figure 6-14. Using E-Extender

AS075.0

## Notes:

By simply using the file BIFs, **%found**, **%eof**, **%equal**, and **%open**, in your programs, you are circumventing potential problems.

Now we want to focus your attention on **%Error**, which requires the use of the **E** opcode extender with the opcodes that are supported.

# Using the (E) extender

- (E)rror Opcode extender:
  - Used with a limited range of operations
  - Most file-handling operations: READ, CHAIN, SETLL, and so forth
  - Also used with CALLP, IN, DSPLY, TEST, and so forth
- %ERROR:
  - Returns \*On or \*Off
  - Value is set to \*Off before the operation begins
  - Normally used with %Status to determine the exact cause
- %STATUS:
  - Returns 0 if no error has occurred since the beginning of the last operation that specified the (E) extender
  - %STATUS(file) returns the same value as the \*STATUS field in INFDS
  - %STATUS returns most recent value set for program or file status

© Copyright IBM Corporation 2009

Figure 6-15. Using the (E) extender

AS075.0

## Notes:

The visual summarizes how **%error** and **%status** work in conjunction with the E-extender for opcodes.

There are other means of handling errors that we do not discuss in this class, but, RPG IV provides five ways to code HLL-specific handlers and to recover from an exception:

1. (E) operation code extender: recommended
2. Monitor group
3. ILE Condition Handler: discussed in AS10/AS100
4. INFSR error subroutine: not recommended
5. \*PSSR error subroutine: not recommended

The highlighted methods are the ones that we recommend with your code. You might experience the other methods when maintaining legacy applications. Modify the code to use the newer techniques wherever possible.

When you are using the (E) extender, the relevant program and file error information can be obtained by using the **%STATUS** and **%ERROR** built-in functions.

# Using BIFs with opcodes

IBM i

	Built In Function		
Opcode	Record Found?	I/O Error?	BOF? EOF?
Chain	%found	%error	
Read		%error	%eof
ReadC		%error	%eof
ReadE		%error	%eof
ReadP		%error	%eof
ReadPE		%error	%eof
SetLL	%found	%error	%equal
SetGT	%found	%error	
Write		%error	%eof
ExFmt		%error	
Update		%error	
Delete	%found	%error	
Open		%error	
Close		%error	

© Copyright IBM Corporation 2009

Figure 6-16. Using BIFs with opcodes

AS075.0

## Notes:

This visual summarizes the options available to you for error handling using BIFs with RPG IV operation codes.

You have already been using some of the available BIFs or indicators to detect problems that you can handle in your code, such as **Record Not Found**.

Coding the **E**-extender and **%ERROR** BIF is **optional**. The system handles all errors that you do not handle using the default error handler.

To obtain valid results from the **%ERROR** BIF, you must use it in conjunction with the **E** opcode extender.

**%ERROR** returns a character value of **1** if the most recent opcode with the **E**-extender resulted in an error condition.

Before an operation with extender **E** specified executes, **%ERROR** is set to a character value of **0** and does not change following the operation unless an error occurred.

Trapping the exception is not enough. You need more information to determine the exact nature of problem. Such information is available from system feedback areas which are

accessible through RPG IV data structures (INFDS and PSDS) and from the **%STATUS** BIF.

# Using %Error to handle an error

IBM i

```

000001    FEmpMast  UF   E          K Disk
000002
000003    /FREE
000004      Chain(E) EmpNumber EmpMast;
000005
000006>>■  IF %Error;
000007    // We have a problem - execute error-handling routine
000008    Exsr ErrorSR;
000009  Endif;
000010
000011  *InLR = *on;
000012
000013  Begsr ErrorSR;
000014  // Need to determine exact cause of problem and take appropriate action
000015  :
000016  // Need more information ....
000017  :
000018  // Branch back to mainline code
000019  /FREE
000020  Endsr;
000021
000022  /END-FREE

```

© Copyright IBM Corporation 2009

Figure 6-17. Using %Error to handle an error

AS075.0

## Notes:

The **E** opcode extender is used for basic error handling. The **E-extender** is used in conjunction with the **%ERROR BIF** which has a value of ‘1’ if an error condition is encountered by the opcode with the **E-extender** included.

We know we have an error, but:

1. How should we handle it?
2. What other information about the error can we obtain to help us to handle the error the best that we can?

## %ERROR and E

Always use the **E-extender** with I/O opcodes with a **%ERROR BIF** coded directly below. It is a good idea to code the opcode with the **E-extender** and the **%Error BIF** directly beneath it. If you forget, the default error handler displays a message from which you will be unable to recover. Or, more likely, **%ERROR BIF** is set by the *most recently executed opcode with the E-extender included*.

# INFDS: File information data structure

				K DISK	INFDS(EmpMastDS)
000001	FEmpMast	UF	E		
000004					
000005	D EmpMastDS		DS		
000006	D FileName		*FILE		
000007	D Open			1	Overlay(EmpMastDS:9)
000008	D EOF			1	Overlay(EmpMastDS:*Next)
000009	D Error		*STATUS		
000010	D Opcode		*OPCODE		
000011	D Routine		*ROUTINE		
000012	D ListingLine			7	Overlay(EmpMastDS:30)
000013	D Record		*RECORD		
000014	D MsgID			T8	Overlay(EmpMastDS:46)
000015					
000016					
000017					

- \*File = first 8 characters of filename
- Retrieve full filename from pos 83-92 of InfDS

© Copyright IBM Corporation 2009

Figure 6-18. INFDS: File information data structure

AS075.0

## Notes:

The file information data structure (known as the **INFDS**) contains useful data for exception handling in a specific format that you can use in your RPG IV programs.

This data is described in a data structure. The data can be retrieved using length or from and to positions. Certain subfields are redefined using special RPG IV names beginning with an asterisk (\*).

The INFDS provides information to your program about the file and I/O operations to the file. A file information data structure can be uniquely linked to each file used by your program.

This file information data structure example provides us with the field **Error**, and many other fields, that can be used to fetch the error status code for the file.

There is also the **%Status** BIF that retrieves the value of the error status code (like the **\*STATUS** field of the INFDS) for the file. **%Status(filename)** is set regardless of whether you have coded the INFDS in your program.

Note the relationship between **EmpMast** and a file informational data structure, **EmpMastDS** is made on the F-specification. The requested information is defined on the D-specifications wherein program subfields are assigned values from the system's file, open and I/O feedback areas. These values are assigned by using the system keywords or from and to positions.

The *RPG IV Reference Manual* is helpful in locating some of the more commonly used subfields as well as detailing from/to positions, data format, field length and keyword, if any.

The layout of the data in the **INFDS** includes:

- A standard layout area that applies to all file types:
  - File Feedback (length is 80)
  - Open Feedback (length is 160)
  - Input/Output Feedback (length is 126)
- A variable layout portion depending on file type:
  - Device Specific Feedback
  - Get Attributes Feedback

Some areas of the INFDS are updated for every I/O operation. You can use the opcode, **Post**, to refresh the File Information Data Structure.



### Note

---

The **\*FILE** keyword retrieves only the first eight characters of the file name. It is much better to retrieve the full file name from positions 83-92 of the INFDS.

Finally, think about making the INFDS a copy member of an externally described data structure. Note that we show positional notation. You could use length notation and **OVERLAY** if you prefer.

## Subset of file status codes

IBM i

<u>Code</u>	<u>Description</u>
00000	Operation successful
00013	Subfile full
01021	Duplicate key – not allowed
01041	Array/Table sequence error
01211	I/O operation to closed file
01215	OPEN issued to file already opened
01218	Unable to allocate record
01221	Update without prior READ
01241	Record number not found
01331	Wait time exceeded for READ from WORKSTN file

© Copyright IBM Corporation 2009

Figure 6-19. Subset of file status codes

AS075.0

### Notes:

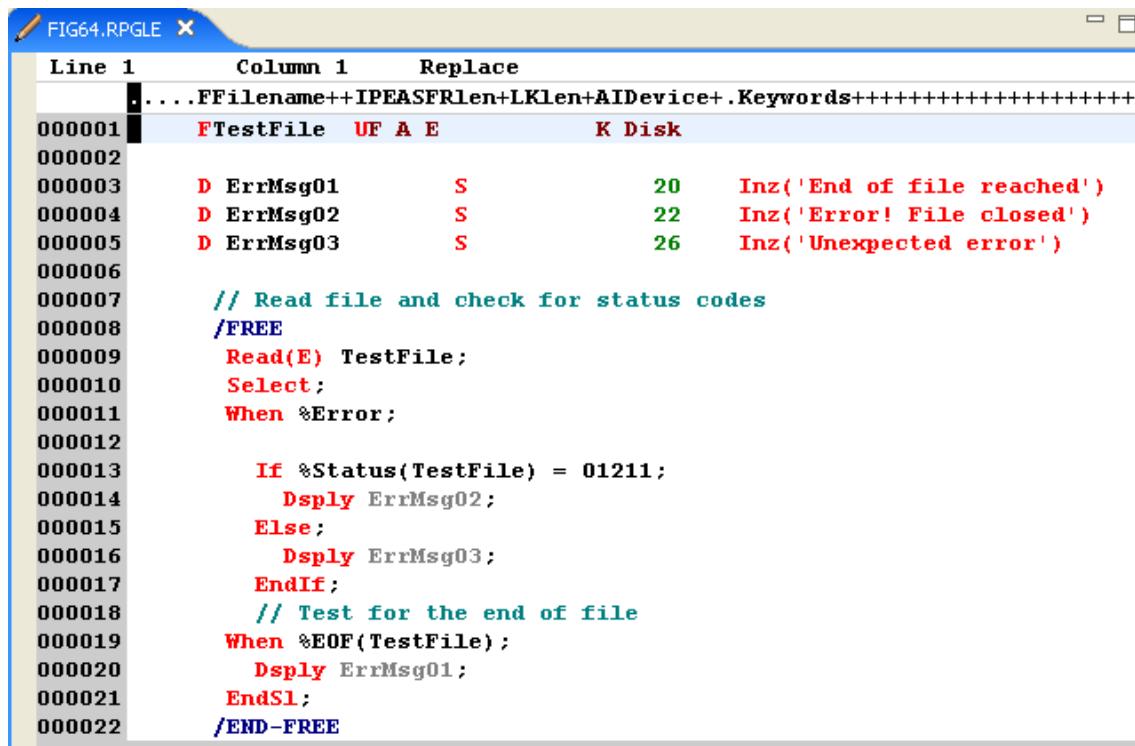
This is a list of some of the commonly used status codes from the file information data structure.

For example, a status of **00000** indicates that the preceding file I/O operation completed normally. *Any file status code with a value of 00100 or greater is considered an exception.* If you coded the E-extender for an opcode with %error, %error would be set to '1' for a status code of 00100 or greater that applies.

All values of the file status code can be retrieved using the **%Status** Built-in Function.

# E-extender, %Error, and %Status

IBM i



```

FIG64.RPGL X
Line 1      Column 1      Replace
.....FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords+++++
000001 FTestFile UF A E K Disk
000002
000003 D ErrMsg01 S 20 Inz('End of file reached')
000004 D ErrMsg02 S 22 Inz('Error! File closed')
000005 D ErrMsg03 S 26 Inz('Unexpected error')
000006
000007 // Read file and check for status codes
000008 /FREE
000009   Read(E) TestFile;
000010   Select;
000011   When %Error;
000012
000013     If %Status(TestFile) = 01211;
000014       Dsply ErrMsg02;
000015     Else;
000016       Dsply ErrMsg03;
000017     EndIf;
000018     // Test for the end of file
000019     When %EOF(TestFile);
000020       Dsply ErrMsg01;
000021     EndSL;
000022   /END-FREE

```

© Copyright IBM Corporation 2009

Figure 6-20. E-extender, %Error, and %Status

AS075.0

## Notes:

In this example, we are using several BIFs to determine what is the problem when we encounter an error. Notice that:

- **%Error** is set to '1' if an error occurs.
- To determine the specific error, we use the **%Status** BIF. **%Status** is used to fetch file and program status codes that tell you whether something went wrong and what the error was. **%Status** is set by file and program status code errors.
- In this case, we test for end of file using the **%Eof** BIF. Always use the specific file name you are testing with **%Eof**. The **%Eof** is set as a the result of the latest read or write. If you do not specify the file name, you may get a result for another file that you did not expect.

To enable the **(E)** operation code extender handler, you code an **(E)** or **(e)** with any of the operation codes that support the e-extender. They are listed in the *ILE RPG IV Reference Manual*.

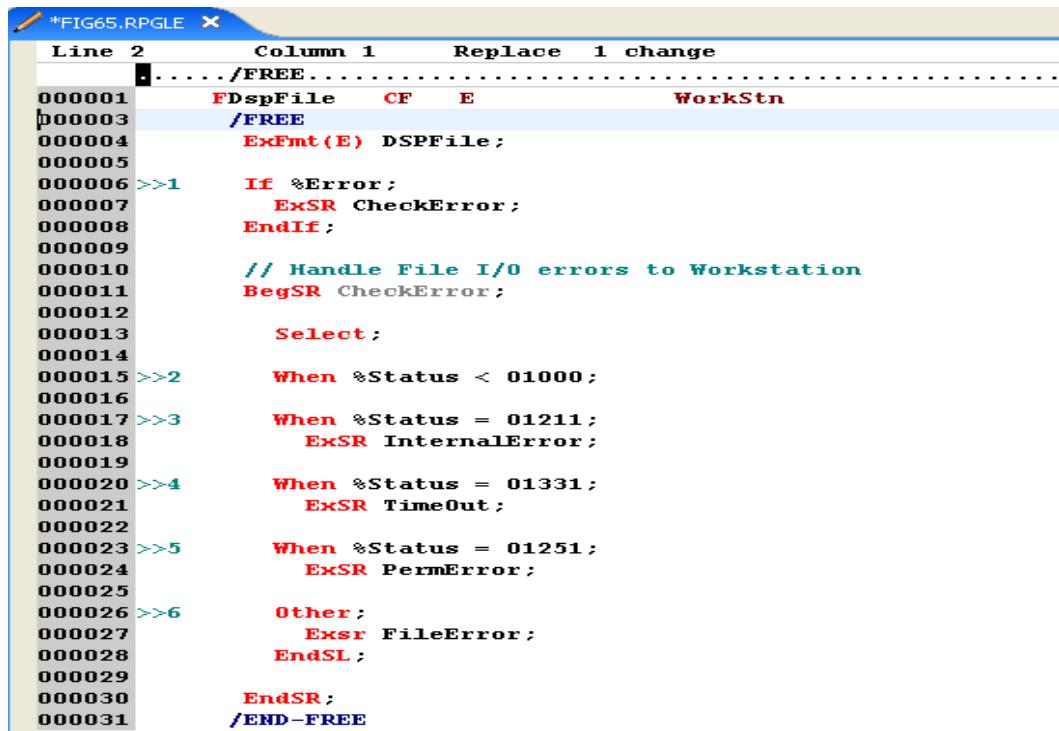
Coding the **(E)** extender affects the value returned by the built-in functions **%ERROR** and **%STATUS** for exceptions. You can then use these built-in functions to test the returned values and determine what action to take.

If an '**E**' operation code extender is present on the calculation specification and the exception is one that is expected for that operation:

- The return values for the built-in functions **%STATUS** and **%ERROR** are set. Note that **%STATUS** is set when any exception occurs even if the '**E**' extender is not specified.
- The exception is handled.
- The program resumes.

# %Status with WorkStn file

IBM i



```
*FIG65.RPGL X
Line 2      Column 1      Replace 1 change
...../FREE.....  

000001    FDspFile   CF   E           WorkStn  

000003    /FREE  

000004    ExFmt(E)  DSPFILE;  

000005  

000006 >>1    If %Error;  

000007        ExSR CheckError;  

000008    EndIf;  

000009  

000010    // Handle File I/O errors to Workstation  

000011    BegSR CheckError;  

000012  

000013    Select;  

000014  

000015 >>2    When %Status < 01000;  

000016  

000017 >>3    When %Status = 01211;  

000018        ExSR InternalError;  

000019  

000020 >>4    When %Status = 01331;  

000021        ExSR TimeOut;  

000022  

000023 >>5    When %Status = 01251;  

000024        ExSR PermError;  

000025  

000026 >>6    Other;  

000027        Exsr FileError;  

000028    EndSL;  

000029  

000030    EndSR;  

000031    /END-FREE
```

© Copyright IBM Corporation 2009

Figure 6-21. %Status with WorkStn file

AS075.0

## Notes:

The **%STATUS** built-in-function:

- Can return the most recent value set for the program status or file status.
- Is set every time the program status or *any* file status changes, (usually when an error occurs). Therefore, it is important that you know exactly what you are testing when you code the %status BIF at various points in your code.
- Returns the file status of **DSPFILE** most recently changed by an I/O operation when **%Status(DSPFILE)** is explicitly coded. When a file name is specified, the value contained in the INFDS “**STATUS**” field for the specified file is returned. The INFDS does not have to be specified for the file unless there is other information you want to obtain from it.
- Returns the file status of the file most recently operated on or the status of the program, when **%Status** is coded without a file name parameter. Which value is returned depends on when the **%Status** is encountered during execution.

- Is initialized with a return value of zeros and is reset to zeros before any operation, in which with an **E-extender** is specified, executes.
- Is best checked *immediately* after an operation with the **E-extender** specified.

In this visual, we are handling various I/O errors to the workstation using the **%Status** BIF:

1. Note that immediately after testing whether **%Error** has been set to ‘1’, (1), we execute a subroutine to determine the type of error and based on the error execute another subroutine. Even though **%Status** is used without the file\_name parameter, the value tested is always the file status code for the workstation file.

In this program, we are testing for these errors using a **SELECT** group:

1. When **%Status < 01000**, the operation was successful.
2. When **%Status = 01211**, we tried to write to a file that was not open.
3. When **%Status = 01331**, we exceeded the wait time for the READ performed by EXFMT.
4. When **%Status = 01251**, we experienced a permanent I/O error.
5. The **OTHER** catches all other I/O errors.

# Handling a duplicate key on write to file

IBM i

```

000001    FEmpMast   UF A E      K Disk
000002    FTrans     IF  E      Disk
000003    FDupFile   O   E      Disk
000004
000005    /FREE
000006        Read Trans;
000007        DoW not %eof(Trans);
000008
000009
000010        Write(F) EmpMast;
000011        If %error and %status(EmpMast) = 1021;
000012            Write Dupfile;
000013        Else;
000014
000015            // additional code
000016        EndIf;
000017
000018        Read Trans;
000019    EndDo;
000020
000021    *InLR = *on;
000022    /END-FREE

```

© Copyright IBM Corporation 2009

Figure 6-22. Handling a duplicate key on write to file

AS075.0

## Notes:

The **E-extender** is coded with the WRITE operation. The **%Error** BIF is set to ‘1’ whenever the write does not complete normally, and set to ‘0’ when it is successful. However, the fact that **%Error** is set to ‘1’ is not enough to determine that there is a duplicate record in the file.

To be certain of the source of the exception, the program also uses the **%Status** BIF to test the value of the file status code.

If the value of **%Status** is 1021, a duplicate key was found and the program writes the record to a duplicate key file. In this example, the key must be unique.

**Note**

It might be better still to avoid the duplicate key error altogether. You could use **SETLL** (and **%Equal**) to check the existence of a duplicate key before you attempt the WRITE. Both methods workaround the possible error. Doing this not only avoids the error altogether, but it also eliminates the need for a READ and is more efficient. Which technique you choose depends on the data. If there is a large proportion of duplicates, then check each record first; if the proportion of duplicates is small, then rely upon your error handling to deal with them.

# Using SETLL to avoid duplicate key error

IBM i

```

000600     Read Trans;
000602     DoW not %eof(Trans);
000603     IF TranType = 'A';
000604         ExSr AddRecord;
000605     EndIF;
000606
000607     IF TranType = 'C';
000608         ExSr ChgRecord;
000609     EndIF;
000610
000611     IF TranType = 'D';
000612         ExSr DltRecord;
000613     EndIF;
000614
000615         // additional code
000616
000617     Read Trans;
000618 EndDo;
000619
000620     *InLR = *on;
000621
000622 BegSr AddRecord;
000800>>     SetLL EmpNum EmpMast;
000900>>     IF not %Equal(EmpMast);
001000         Write(E) EmpMast;
001100         IF %error;
001200             ExSr OtherError;
001300         EndIF;
001400     EndIF;
002000 EndSr;

```

© Copyright IBM Corporation 2009

Figure 6-23. Using SETLL to avoid duplicate key error

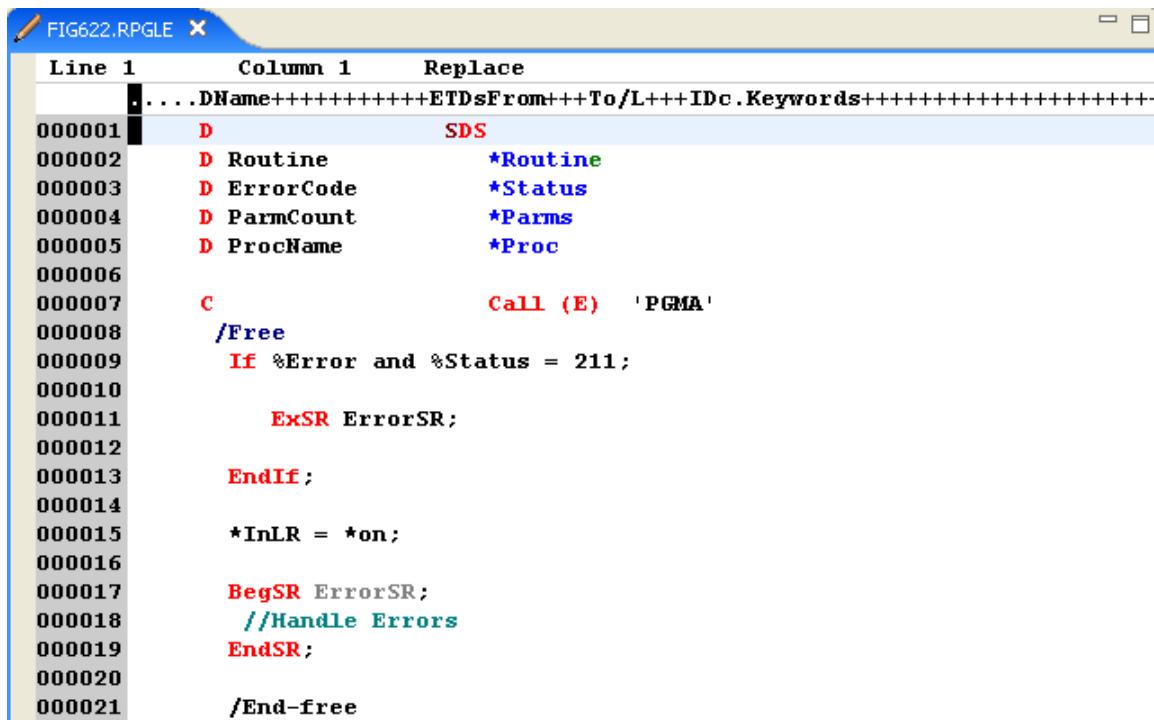
AS075.0

## Notes:

We modified the previous visual to show you how you might avoid a duplicate key error. Instead of writing code that handles the error, **SetLL** positions the cursor without performing a read or a write. The **%Equal** BIF checks whether the search key from the transaction file (**EmpNum**) is equal to an existing key in the file when performing an add transaction. The same logic could be used to check whether a key for the record already exists for a change or a delete transaction to avoid an error in these situations as well.

# Program status data structure

IBM i



The screenshot shows a code editor window titled 'FIG622.RPGLE'. The code is a RPGLE program. Column 1 shows line numbers from 000001 to 000021. Column 2 shows the input specification (D for Data, C for Call, etc.). Column 3 shows the replace field, which includes system names like \*ROUTINE and \*STATUS.

```

Line 1      Column 1      Replace
.....DName++++++ETDsFrom+++To/L+++IDc .Keywords+++++
000001      D          SDS
000002      D Routine    *Routine
000003      D ErrorCode  *Status
000004      D ParmCount *Parms
000005      D ProcName   *Proc
000006
000007      C          Call (E) 'PGMA'
000008      /Free
000009      If %Error and %Status = 211;
000010
000011      ExSR ErrorSR;
000012
000013      EndIf;
000014
000015      *InLR = *on;
000016
000017      BegSR ErrorSR;
000018      //Handle Errors
000019      EndSR;
000020
000021      /End-free

```

© Copyright IBM Corporation 2009

Figure 6-24. Program status data structure

AS075.0

## Notes:

Let us quickly review the PSDS with which many are familiar.

In the same way that IBM i provides information about each file used by a program, it also provides information about the program itself.

This information can also be retrieved using a special data structure, the Program Status Data Structure. This data structure does not relate to any particular file. Instead, it provides information about the processing being done by the program.

The program identifies this as a program status data structure by specifying an S in column 18 on the Input Specification that contains the DS entry (1).

Like the File Information Data Structure, the locations of certain frequently used information fields have been associated with reserved system names that can be specified in place of their FROM and TO positions:

- **\*ROUTINE** retrieves the name of the called RPG routine in which the error occurred.
- **\*STATUS** has the same meaning as it does for files, although as we see, the values of the status codes returned are different.

- **\*PARMS** is a 3-digit whole number that contains a count of the number of parameters passed to this program by the calling program.
- **\*PROC** is a character field 10-positions long, containing the name of the procedure (often the program name) in which this program status data structure is specified.

A Program Status Data Structure can provide information about the program itself and about exceptions or errors that occur during the program's execution. Like the File Information Data Structure, a program status data structure has predefined subfields that are accessible by keyword or from/to positions.

This example uses the **E** extender and the **%Error** BIF with the CALL operation.

While we show you how to code the **\*STATUS** field in the program status data structure, we do not reference it in the calculations. We actually do not need to have coded it at all as **%Status** fetches the value for us. The **%Status** BIF is set whether or not you have coded the PSDS in your program.

If the program attempts to run the CALL opcode and an error occurs, **%error** will return a value of '1'. If the value of **%Status** equals 211, the program, **PGMA**, was not found and a subroutine named **ErrorSR** is executed.

But, even though we have gracefully identified the error and executed a subroutine:

- What else should we be doing after the subroutine has completed execution?
- Where should the program continue processing?

The subroutine to handle the error could be any name you want.

### References:

*ILE RPG for Reference Manual, Chapter 5, File and Program Exception/Errors*

## Examples of program status codes

IBM i

<u>*STATUS</u>	<u>Explanation</u>
00121	Invalid Array Index
00202	CALLED Program Failed
00211	CALLED Program not Found
00231	CALLED Program Halt Indicator On
00333	Error on DISPLAY operation
00414	Not Authorized to Use Data Area
00415	Not Authorized to Update Data Area
00431	Data Area Previously Locked
00907	Decimal Data Error
00999	System Program Exception Error

© Copyright IBM Corporation 2009

Figure 6-25. Examples of program status codes

AS075.0

### Notes:

Any program status code of **0100 or greater** is considered to be an exception. If you coded the E-extender with %Error, %Error would be set to '1' for any of the above codes that apply.

# What else is in the program status data structure

IBM i

<u>From / To</u>	<u>Field Contents</u>
16 - 20	Previous Status Code
21 - 28	RPG IV Source Sequence Number
40 - 46	Exception Error ID (MCH or CPF)
201 - 208	Name of File Last Used
244 - 253	Job Name
254 - 263	User Name
264 - 269	Job Number
276 - 281	System Date in UDATE format
288 - 293	Date Compiled
304 - 333	Source File/Library/Member

D	SDS		
D Stmt		21	28
D PgmError		40	46
D UserId		254	263

© Copyright IBM Corporation 2009

Figure 6-26. What else is in the program status data structure

AS075.0

## Notes:

As in the case of the File Status Data Structures, much more information is available than what can be retrieved using the reserved system words. Notice that both exception error IDs and the complete job name, including the User Name, are available to the program. Some of the other information shown here can be very useful when changing or debugging a program.

To retrieve this additional information, FROM and TO positions must be coded on the D-Specifications for the data structure. If desired, these entries can be mixed with ones that use the reserved system words.

# Error handling recommendations

IBM i

- You can never add too much error handling code.
- The process is iterative.
- Tools and techniques recommendations:
  - Use E-extender and %Error
  - Use %Status
  - Use PSDS and INFDS for specific information
  - Code error handling and recovery in subroutines
  - Code %Error and %Status immediately after operation and before other I/O BIFs
- Available BIFs focus heavily on I/O opcodes.
- Use monitor groups.
- Implement ILE condition handlers as needed.
- But, always check for errors *before* they happen:
  - Zero divisor
  - Numeric overflow
  - Decimal data error
  - Array index zero, negative, or greater than number of elements

© Copyright IBM Corporation 2009

Figure 6-27. Error handling recommendations

AS075.0

## Notes:

We have talked about a number of ways your code can handle errors to avoid having the default error handler that you are taking care of the problem.

However, many of these techniques (mostly BIFs) apply mainly to the I/O opcodes and a few others (CALL and IN, for example). There are no similar BIFs for EVAL, FOR, DOW, DOU and other free form opcodes that support free form expressions.

The best plan for handling errors is to *anticipate them and avoid them in your programs*. Check for a zero divisor. Make sure that your result fields are large enough to hold the result of the expression.

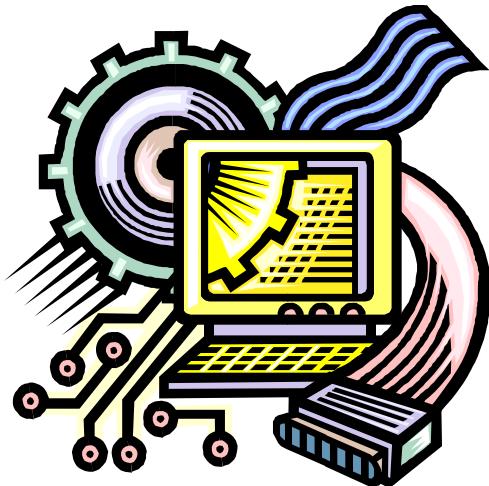
When coding error handling BIFS (%Error and %Status), be sure that your error handling logic is processed before any other I/O BIFs.

If %Error is true, the value of the %Found, %Equal, and %Eof BIFs should be ignored.

Finally, error handling is a never ending process. There is always more to learn and more error handling code to add.

## Machine exercise: Error-handling BIFs

IBM i



© Copyright IBM Corporation 2009

Figure 6-28. Machine exercise: Error-handling BIFs

AS075.0

### Notes:

Perform the machine exercise “Error-handling BIFs.”

## 6.2. Monitor groups

# What can be coded to handle this error?

IBM i

```

FIG628.RPGLE X
Line 1      Column 1      Replace
.....FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords+++++
000001     FApInv_PF  if   e           Disk
000002     D Keyarr        S           6S 0 Dim(5)
000003     D Index         S           1S 0 INZ(0)
000004     /Free
000005     Chain(e) Keyarr(index) apinv_pf;
000006
000007     If %Error;
000008       Dsply 'Index Error';
000009     EndIf;
000010
000011     If not %Found(apinv_pf);
000012       Dsply 'Not Found';
000013     Endif;
000014
000015     *InLR = *On;
000016 /End-free

```

The code segment at line 5, 'Chain(e) Keyarr(index) apinv\_pf;', is circled in red.

© Copyright IBM Corporation 2009

Figure 6-29. What can be coded to handle this error?

AS075.0

## Notes:

Look at this example carefully. The programmer coded the error handling techniques that we have discussed. But, when the program is executed, an error occurs:

Message ID . . . . . : RNQ0121

Message . . . . : An array index is out of range (C G D F).

Cause . . . . . : RPG procedure FIG627 in program AS07000/FIG627 tried to use an array index at statement 17 which was less than one or greater than the number of elements in the array.

The message is displayed because the value of the Index is 0. We could have checked for a value of zero, but what if we have a value greater than the number of elements in the array?

How can we handle this type of error?

## Monitor group (1 of 2)

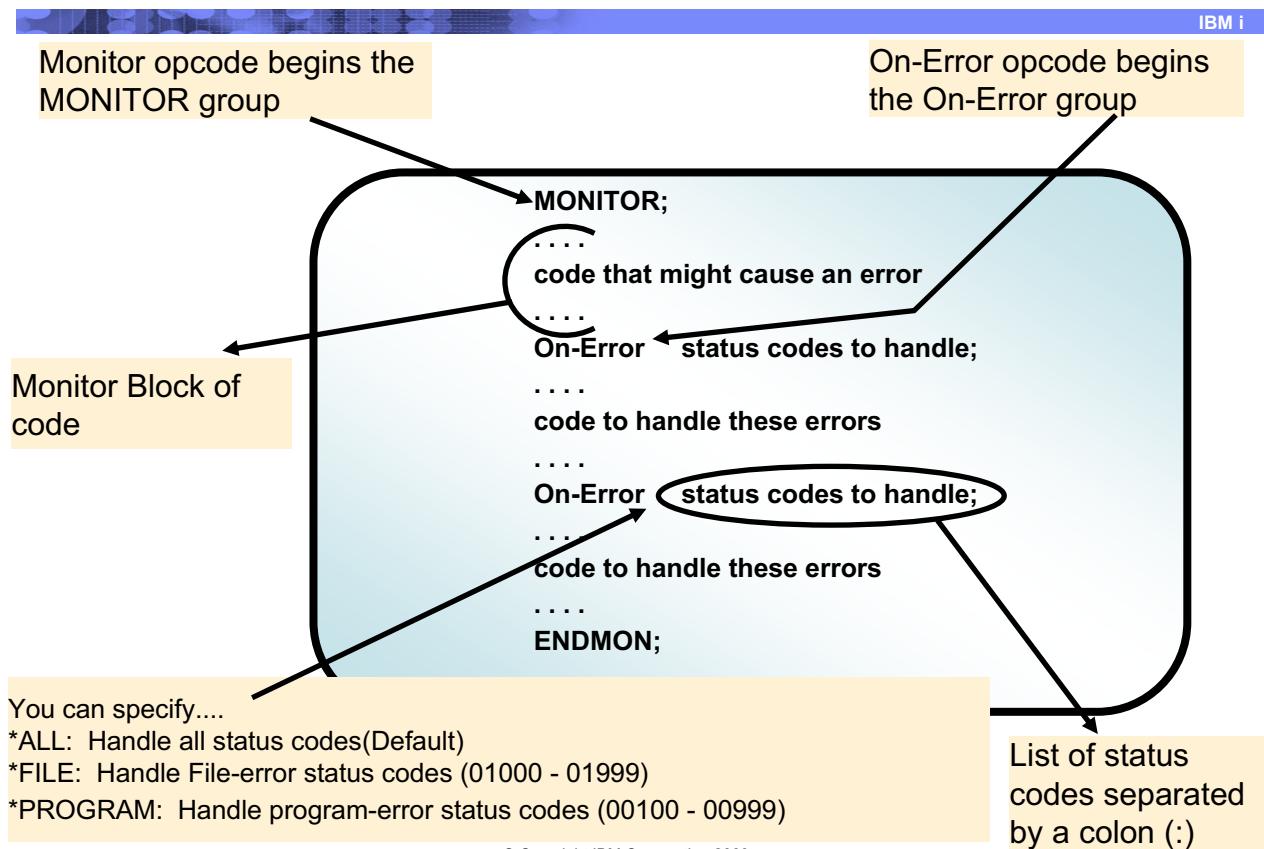


Figure 6-30. Monitor group (1 of 2)

AS075.0

### Notes:

Notice that the code you want to monitor must be coded within the Monitor Group, that is, between the **Monitor** and the **EndMon** statements.

The errors that you can handle in an **On-error** are:

**Specific status codes - One or more status codes (separate each code with a colon).**

- \***File:** File-error codes
- \***Program:** Program-error status codes
- \***All:**
  - Any and all status codes
  - Useful as a catch-all

## Monitor group (2 of 2)

IBM i

The MONITOR block consists of the READ statement and the IF group.

'1211' is issued if file is  
not opened

We could  
have coded  
\*ALL

If no errors occur, control  
passes to ENDMON

```

CODE - <AECAUX>AS07V2LIB/QRPGLESRC(FIG630)
File Edit View Actions Options Windows Help
Row 1 Column 1 Replace
000001 /Free
000002 Monitor;
000003
000004 Read File1;
000005 IF Not %EOF(File1);
000006 Line = %Subst(Lin(i) : %Scan('***' : Line(i)) +1);
000007 EndIf;
000008
000009 On-Error 1211;
000010 // ... handle file-not-open
000011 On-Error *File;
000012 // ... handle other file errors
000013 On-Error 00100:00121;
000014 // ... handle string/array-error
000015 On-Error
000016 // ... handle all other errors
000017 Endon;
000018 /End-free
IBM Live Parsing Editor

```

Figure 6-31. Monitor group (2 of 2)

AS075.0

### Notes:

Remember, in RPG IV, there are two native constructs for handling exceptions. You can use the error indicator or **E-Extender**, which enables handling of a predefined set of exceptions.

To enable even more control of exception handling within your RPG IV programs, the **MONITOR** opcode (or group) is available. It consists of a MONITOR block, one or more **ON-ERROR** blocks, and an **ENDMON** operation.

The **MONITOR** group contains the code being monitored for. In this example, you see the **READ** operation code being placed in the **Monitor** group. All code between the **Monitor** and the **EndMon** statements are checked for errors. When an error occurs, the **On-Error** blocks are checked until the error has been handled.

The **On-Error** blocks test for certain status codes or a range of status codes. If any one test evaluates to true because an exception has occurred, the **On-Error** block is executed and the error is handled in the program. If the error is not handled by the **On-error** block, the error is passed to the next lower level in the error handling hierarchy.

When an error has been detected and is handled by an **On-error** within the **MONITOR** group, control passes to the statement following the **ENDMON** statement.

And, if all the statements within the **MONITOR** group are processed without errors, control passes to the statement following the **ENDMON** statement.

# Monitor group isolates error (1 of 2)

```

FIG631.RPGLE X
Line 1      Column 1      Replace
000001      .....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8....+....9.
000002      FApInv_Pf  if   e           Disk
000003
000004      D Keyarr       S           6S 0 Dim(5)
000005      D Index        S           1S 0 Inz(0)
000006
000007      /Free
000008
000009 >>      Monitor;          // Beginning of monitor group
000010
000011      Chain(e) Keyarr(Index) ApInv_Pf;
000012
000013 >>      On-Error 00121;        // Check for a specific error
000014 >>      Dsply 'Index Error';
000015 >>      *InLr = *On;
000016 >>      Return;
000017
000020 >>      Endmon;          // End of Monitor Group
000021
000022
000023      If Not %Found(ApInv_Pf);
000024      Dsply 'Not Found';
000025      Endif;
000026
000027      *InLr = *On;
000028
000029      /End-Free
000030

```

© Copyright IBM Corporation 2009

Figure 6-32. Monitor group isolates error (1 of 2)

AS075.0

## Notes:

When we run the program this time, the array index error is trapped in the program and a message is displayed.

The statements marked >> are new or changed.

## Monitor group isolates error (2 of 2)

```

SUDLE>CALLAOA>AS0750>URPGLE>SRC(TESTR)
File Edit View Actions Options Windows Help
Row 1 Column 1 Replace
-----DName+++++ETDsFrom+++To/L+++IDc.Keywords+++++Comm
000500 D Result S 10A
000600 D ErrorFlag S N
000700 DErrorMsg S 30A
000800
000900 D Number S 5P 2 Inz(123.45)
001000 D Ten S 3P 0 Inz(10)
001100 D Zero S 3P 0 Inz(0)
001200 D MsgQ C Const('*REQUESTER')
001300
001400 /Free
001500
001600 // Normal division
001700 Eval ErrorMsg = 'Normal Division ' + %Char(Number) + ' / '
001800 + %Char(Ten);
001900 Dsply ErrorMsg MsgQ;
002000 Result = %EditC(Number / Ten : 'J');
002100 Dsply Result MsgQ;
002200
002300 Monitor
002400 // Divide by zero error condition
002500 Result = %EditC(Number / Zero : 'J');
002600
002700 On-Error 00702: // Divide by zero
002800 Eval ErrorMsg = 'Warning ' + %Char(%Status);
002900 Dsply ErrorMsg MsgQ;
003000 Result = 'Div by 0';
003100 EndMon;
003200 Dsply Result MsgQ;
003300
003400 Monitor;
003500 // Numeric overflow error condition
003600 Number = Number * Ten;
003700 Result = %EditC(Number : 'J');
003800
003900 On-Error *Program: // Any other error
004000 Eval ErrorMsg = 'Serious Error (overflow) ' + %Char(%Status);
004100 Dsply ErrorMsg MsgQ;
004200 Result = 'Any error';
004300 EndMon;
004400 Dsply Result MsgQ;
004500
004600 *InLR = *On;
IBM Live Parsing Editor

```

© Copyright IBM Corporation 2009

Figure 6-33. Monitor group isolates error (2 of 2)

AS075.0

### Notes:

The MONITOR/ENDMON block contains the code being monitored for.

The ON-ERROR blocks test for certain status codes or a range of status codes. If any one test evaluates to true (an exception does occur), the ON-ERROR block is executed and the error is handled in your program.

In this example, we are checking for a divide by zero condition. We use a status code to trap zero divide conditions. As well, we use a generic catch all to trap all other program errors. We know that the only other possible error would be numeric overflow.

# Error handling hierarchy

IBM i

1. E-extender / %Error (or error indicator)
2. Monitor group
3. ILE Condition Handler
4. Error subroutine
  - a. I/O Error Subroutine (INFSR for file errors)
  - b. Program Error subroutine (\*PSSR for all other errors)
5. Default Error handler



© Copyright IBM Corporation 2009

Figure 6-34. Error handling hierarchy

AS075.0

## Notes:

The Monitor Group is next in the error handling hierarchy immediately after the system checks for an error indicator (not recommended) or an E-extender to handle an error.

If a monitor group is present for the operation in error, control is passed to the on-error section of the Monitor Group.

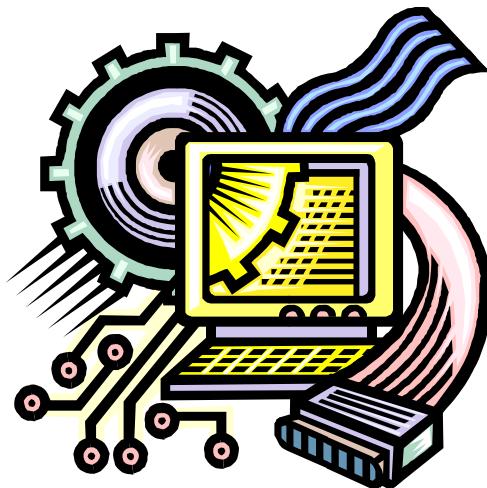
When an On-error condition is satisfied and the error is handled, control is passed to the EndMon statement of the monitor group and execution continues.

We cover ILE Condition Handlers in the course that follows this one--AS10/AS100.

We recommend that you do not use Error Subroutines.

# Machine exercise: Using monitor groups

IBM i



© Copyright IBM Corporation 2009

Figure 6-35. Machine exercise: Using monitor groups

AS075.0

## Notes:

Perform the machine exercise “Using monitor groups.”

## Unit summary

IBM i

Having completed this unit, you should be able to:

- Use the E opcode extender
- Use the %error, %open, %found, and %status BIFs as part of your error-handling code
- Code monitor groups to manage errors

© Copyright IBM Corporation 2009

---

Figure 6-36. Unit summary

AS075.0

### Notes:

# Unit 7. Processing date and time data

## What this unit is about

This unit teaches the RPG IV programmer how to work with date, time, and timestamp data. Use of the data types is discussed in addition to the conversion of numeric and character fields to the date, time, and timestamp data types.

## What you should be able to do

After completing this unit, you should be able to:

- Code definitions for date, time, and timestamp data types
- Explain the difference between numeric or character date/time fields and the data types DATE and TIME
- Use the %DIFF BIF to calculate date and time durations
- Use %DATE, %TIME, %TIMESTAMP to convert characters to date, time, and timestamp data types
- Use the %SUBDT BIF to extract a subfield from a date, time, and timestamp data type
- Use %YEARS, %MONTHS, %DAYS, %HOURS, %MINUTES, %SECONDS, %MSECONDS to convert a numeric value into a duration
- Explain the date *rounding* process

## How you will check your progress

- Machine exercise

# Unit objectives

After completing this unit, you should be able to:

- Code definitions for date, time, and timestamp data types
- Explain the difference between numeric or character date or time fields and the data types DATE and TIME
- Use the %Diff BIF to calculate date and time durations
- Use %DATE, %TIME, %TIMESTAMP to convert characters to date, time, and timestamp data types
- Use the %SubDt BIF to extract a subfield from a date, time and timestamp data type
- Use %YEARS, %MONTHS, %DAYS, %HOURS, %MINUTES, %SECONDS, %MSECONDS to convert a numeric value into a duration
- Explain the date rounding process

© Copyright IBM Corporation 2009

Figure 7-1. Unit objectives

AS075.0

## Notes:

## What can be done?

IBM i

- Define date (D), time (T), and timestamp (Z) data on D-spec
- Select date and time format (\*MDY, \*EUR, \*HMS)
- Calculate duration between two dates
  - Days = %Diff(DateDue : DateShipped : \*days)
- Extract components of date, time, and timestamp
  - MonthBorn = %SubDt(DateBorn : \*m)
- Convert character data to date and time using %DATE, %TIME, %TIMESTAMP
  - DateToday = %Date(\*date)
- Compare date and time values
- Validate date, time, and timestamp fields using TEST (D/T)
  - Test(DE) CharDate

© Copyright IBM Corporation 2009

Figure 7-2. What can be done?

AS075.0

### Notes:

This visual lists some of the things you can do with date, time, and timestamp data types. These data types make performing date and time-based calculations much easier.

Understanding how to define these types of variables makes it easier for you to use these data types in calculations.

# Defining date, time, and timestamp fields

IBM i

- Define on D-spec or externally using DDS or SQL
- Data types D, T, Z
- Length on D-spec not required
- Format used in program can be specified (default \*ISO)
  - **Timestamp fields:**
    - yyyy-mm-dd-hh.mm.ss.mmmmmm only format
    - Microseconds optional (compiler will pad with zeroes)
- Literals = D'Date value', T'Time value', Z'Timestamp value'

The screenshot shows the IBM i CODE editor interface. The title bar reads "CODE - <05400A>DE85V5LIB/QRPGLESRC(FIG73) \*". The menu bar includes File, Edit, View, Actions, Options, Windows, and Help. A toolbar with various icons is above the main window. The main window displays a table with columns: Row, Column, Replace, and 2 changes. The table rows define fields:

	DName	ETDsFrom	To/L	IDc	Keywords	Comments
000001	D DateFld	\$		D	DatFmt(*DMY)	
000003	D				Inz(D'12/19/03')	
000004	TimeFld	\$		T	TimFmt(*HMS)	
000005	D				Inz(T'13.00.00')	
000006	DTimeStamp	\$		Z	Inz(Z'2003-01-15-12.00.00.000000')	

© Copyright IBM Corporation 2009

Figure 7-3. Defining date, time, and timestamp fields

AS075.0

## Notes:

Date, time, and timestamp data fields can be defined on the D-spec, on the I-spec (for program-described files), or they can be brought in from externally described files. These data types are equivalent to the corresponding DDS data types, except that DDS uses data type 'L' for a date field.

The **INZ** keyword on the D-spec allows you to initialize variables with specific values.

Date, time, and timestamp fields do not require lengths to be specified. Their lengths are implicitly defined based on the data type. If a length is specified, it must be correct for the format. For example, if the format for a date field is \*MDY, the length must be 8 characters, the length of the *edited* field. In the case of date formats, the 2-digit year formats are 8 characters long (except Julian \*JUL which is 6) and the 4-digit formats are 10 characters in length.

The default format for dates and times in the program can be specified on the H-spec. If no format is specified on the H-spec, then the default format for both date and time fields throughout the program is \*ISO. The formats possible are listed on the next visual.

*Literal values* can be entered for dates, times, and timestamps. They must be preceded with D, T or Z with the value that is specified in single quotes. The format of the literal value for the INZ or CONST keywords that are specified on the D-spec *must be* in the default format for the program. This is required even if the format specified on the D-spec is different.

In the example on this visual, the H-spec specifies that the default formats for this program are \*MDY for dates and \*ISO for times. Therefore, those are the formats that are used when specifying the INZ keyword values.

The timestamp data type has only one format of 26 characters in length.

## Date formats

IBM i

Name	Description	Format	Sep	Length	Example
*MDY	Month/Day/Year	mm/dd/yy	/-,&	8	01/15/40
*DMY	Day/Month/Year	dd/mm/yy	/-,&	8	01/01/40
*YMD	Year/Month/Day	yy/mm/dd	/-,&	8	40/01/01
*JUL	Julian	yy/ddd	/-,&	6	94/015
*ISO	Intl. Stds. Organization	yyyy-mm-dd	-	10	1994-01-15
*USA	IBM USA Std.	mm/dd/yyyy	/	10	01/15/1994
*EUR	IBM European Std.	dd.mm.yyyy	.	10	15.01.1994
*JIS	Japanese Ind. Std.	yyyy-mm-dd	-	10	1994-01-15

© Copyright IBM Corporation 2009

Figure 7-4. Date formats

AS075.0

### Notes:

The column labeled **Sep** lists valid separators for the date format. The ampersand (&) symbol indicates a blank is to be used as the separator. The separator character, if different from the default for the format, is specified by a character that follows the format, for example, **\*MDY- (05-10-96)**. Of course, if no separator is specified, the system will use the default separator for the particular format.

Note that these formats are the same as those provided in DDS for externally described Date fields. For those formats with a 2-digit year (\*MDY, \*YMD, \*DMY, \*JUL), the year can be 40 (1940) through 39 (2039); that is, any year less than or equal to 39 is assumed to be in the 21st century. Any year greater than or equal to 40 is assumed to be in the 20th century.

There is a **\*CYMD** format also. The century digit can be **0** for dates through 1999, **1** for dates 2000 to 2099, **2** for dates 2100 to 2199, through **9** for dates 2800 to 2899. Making this change in your database date fields allows you to calculate dates into the 21st century (for example, 096/05/10 is May 10, 1996; 198/03/15 is March 15, 2098).

There are three kinds of date data formats, depending on the range of years that can be represented. This leads to the possibility of a date overflow or underflow condition occurring when the result of an operation is a date outside the valid range for the target field. The formats and ranges are as follows:

Digits -Year	Format	Range of Years
2	*YMD, *DMY, *MDY, *JUL	1940 to 2039
3	*CYMD, *CDMY, *CMDY	1900 to 2899
4	*ISO, *USA, *EUR, *JIS, *LONGJUL	0001 to 9999

\*Cxxx formats **cannot** be declared or stored. These formats can only be used on MOVEs and TEST opcodes, and are for compatibility only.

\*LONGJUL (Long Julian) is similar to \*Cxxx formats except that it is of the form yyyyddd supporting a 4-digit year.

# Date values

IBM i

Name	Description	*LOVAL	*HIVAL	Default Value
*MDY	Month/Day/Year	01/01/40	12/31/39	01/01/40
*DMY	Day/Month/Year	01/01/40	31/12/39	01/01/40
*YMD	Year/Month/Day	40/01/01	39/12/31	40/01/01
*JUL	Julian	40/001	39/365	40/001
*ISO	Intl. Stds. Org.	0001-01-01	9999-12-31	0001-01-01
*USA	IBM USA Std.	01/01/0001	12/31/9999	01/01/0001
*EUR	IBM European Std.	01.01.0001	31.12.9999	01.01.0001
*JIS	Japanese Ind. Std.	0001-01-01	9999-12-31	0001-01-01

© Copyright IBM Corporation 2009

Figure 7-5. Date values

AS075.0

## Notes:

This table is included as a reference of the \*HIVAL, \*LOVAL and default values for the various date formats.

# Time formats

IBM i

Name	Description	Format	Sep	Length	Example
*HMS	Hours:Minutes:Seconds	hh:mm:ss	: . , &	8	14:00:00
*ISO	Intl. Stds. Org.	hh.mm.ss	.	8	14.00.00
*USA	IBM USA Std.	hh:mm AM	:	8	02:00 PM
*EUR	IBM European Std.	hh.mm.ss	.	8	14.00.00
*JIS	Japanese Ind. Std.	hh:mm:ss	:	8	14:00:00

© Copyright IBM Corporation 2009

Figure 7-6. Time formats

AS075.0

## Notes:

The **Sep** column indicates the valid separators for the format. The **&** symbol indicates a blank is used for the separator. The separator character, if you desire one that is different than the default format, is specified as a character that follows the format; for example, **\*HMS&**.

Timestamp fields have a predetermined size and format. Timestamp data must be in the format:

**yyyy-mm-dd-hh.mm.ss.mmmmmmm**

The timestamp data must have a length of 26 characters. Microseconds (**mmmmmm**) are optional for timestamp literals and if not provided, are padded on the right with zeros. Leading zeros are required for all timestamp data.

Note that these formats are the same as those allowed in the database (DDS or SQL) for externally described time fields.

# Time values

IBM i

Name	Description	*LOVAL	*HIVAL	Default Value
*HMS	Hours:Minutes:Seconds	00:00:00	24:00:00	00:00:00
*ISO	Intl. Stds. Org.	00.00.00	24.00.00	00.00.00
*USA	IBM USA Std.	00:00 AM	12:00 AM	00:00 AM
*EUR	IBM European Std.	00.00.00	24.00.00	00.00.00
*JIS	Japanese Ind. Std.	00:00:00	24:00:00	00:00:00

© Copyright IBM Corporation 2009

Figure 7-7. Time values

AS075.0

## Notes:

This table is included as a reference for all the \*LOVAL, \*HIVAL, and default values for the various time formats.

One interesting thing to note is the \*LOVAL and \*HIVAL entries above all mean midnight. When a date is associated with a time (that is, a timestamp), and the date component of two timestamps is the same, the time of 00.00.00 means midnight last night and 24.00.00 means midnight tonight. If you were to calculate the duration between 24.00.00 and 00.00.00, it would equal 24 hours.

The default initialization value for a timestamp field is midnight of January 1, 0001 (0001-01-01-00.00.00.000000).

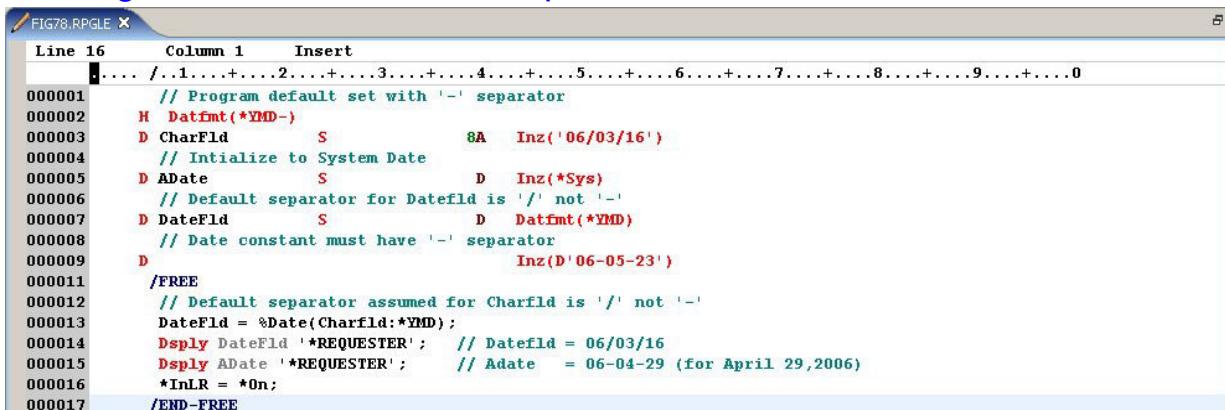
\*HIVAL value for a timestamp = 9999-12-31-24.00.00.000000

\*LOVAL value for a timestamp = 0001-01-01-00.00.00.000000

# Specifying date and time formats

IBM i

- H-Spec:
  - Specify default internal format
  - Default \*ISO
- D-Spec:
  - Overrides H-Spec
  - With INZ or CONST, H-Spec rules
- C-Spec
  - Argument used with TEST opcode and some BIFs



```

FIG78.RPGLE x
Line 16    Column 1    Insert
000001 // Program default set with '-' separator
000002 H DatFmt(*YMD)
000003 D CharFld   S     8A   Inz('06/03/16')
000004 // Initialize to System Date
000005 D ADate      S     D   Inz(*Sys)
000006 // Default separator for Datefld is '/' not '-'
000007 D DateFld   S     D   DatFmt(*YMD)
000008 // Date constant must have '-' separator
000009 D           Inz(D'06-05-23')
000010 /FREE
000011 // Default separator assumed for Charfld is '/' not '-'
000012 DateFld = %Date(Charfld:*YMD);
000013 Dsply DateFld '*REQUESTER'; // Datefld = 06/03/16
000014 Dsply ADate '*REQUESTER'; // Adate = 06-04-29 (for April 29,2006)
000015 *InLR = *On;
000016 /END-FREE
000017

```

© Copyright IBM Corporation 2009

Figure 7-8. Specifying date and time formats

AS075.0

## Notes:

If an H-spec is *not coded* in the program, and a default H-spec is *not found* at compile time, then \*ISO is the default format for the program.

The formats defined on H, D and C are referring to the internal formats that are used in the RPG program. For date and time fields already defined in DDS and brought in as program-described files into the RPG IV program, the external formats should be specified either on the I, O or F-specs. If they are not specified there, the assumption is that the external date/time formats are the same as the default for the program (H-spec or \*ISO).

**Note:** Pay particular attention to the fact that the values that you assign on D-specs must use in the default format for the program, even if you are using a different format for the actual variable.

## Length of Date/Time

Note that date/time have no length specified (on D-spec) because this depends on the format that is chosen. However, this is the *externalized* view of the date. Internally (in a database file), dates are stored as 4-byte values.

Dates are stored in the database as Scalinger numbers. The method of Julian Day Number was introduced by Joseph Scalinger (1540-1609). For more history, browse:

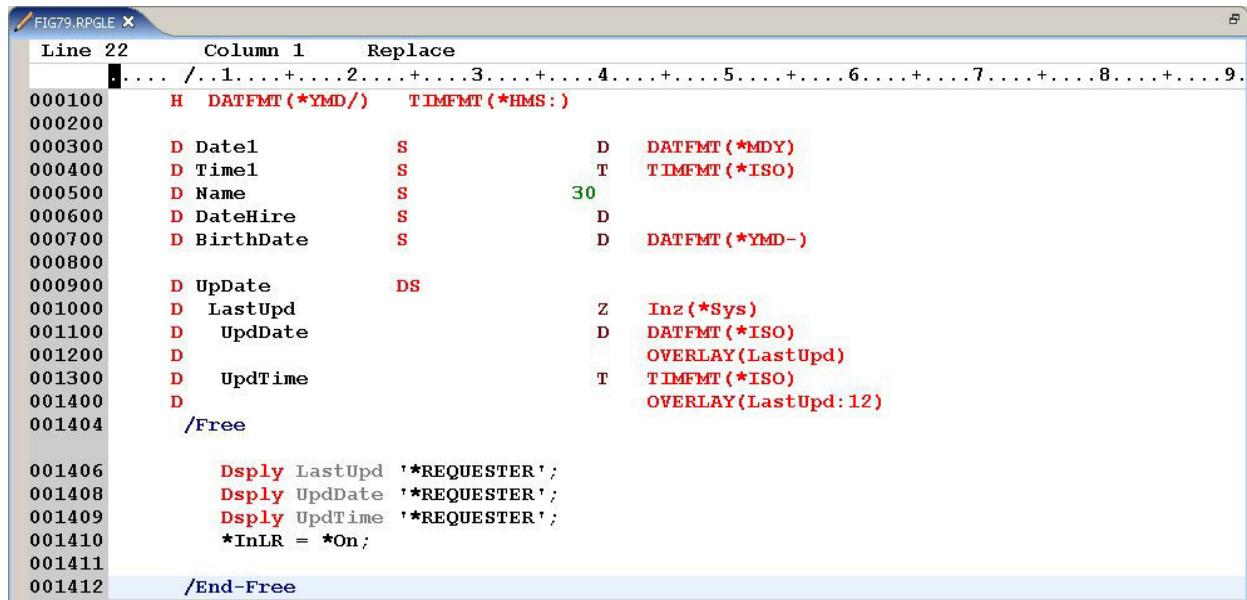
<http://www.friesian.com/numbers.htm>

Times are stored in three bytes with each byte holding two digits in the form hhmmss. Timestamps are stored in 10-byte composite numbers composed of the four byte date, the 3-byte time and the final three bytes containing the microseconds in packed form.

So, the 6/8/10 lengths are the resolved (buffer) lengths only. Even DSPPFM will *expand* date and time fields to a formatted form.

# Define date, time, and timestamp fields

IBM i



```

FIG79.RPGLE X
Line 22    Column 1      Replace
H.... /..1....+....2....+....3....+....4....+....5....+....6....+....7....+....8....+....9.
000100     H DATFMT (*YMD/)   TIMFMT (*HMS:)
000200
000300     D Date1          S           D DATFMT (*MDY)
000400     D Time1          S           T TIMFMT (*ISO)
000500     D Name           S           30
000600     D DateHire       S           D
000700     D BirthDate       S           D DATFMT (*YMD-)
000800
000900     D UpDate          DS
001000     D LastUpd         Z           Inz (*Sys)
001100     D UpdDate         D           DATFMT (*ISO)
001200     D
001300     D UpdTime         T           OVERLAY(LastUpd)
001400     D
001404     /Free
001406     Dsply LastUpd  '*REQUESTER';
001408     Dsply UpdDate  '*REQUESTER';
001409     Dsply UpdTime  '*REQUESTER';
001410     *InLR = *On;
001411
001412     /End-Free

```

What will be the format of DateHire?

© Copyright IBM Corporation 2009

Figure 7-9. Define date, time, and timestamp fields

AS075.0

## Notes:

**BirthDate** is in \*YMD format, but the dash separator (-) is used instead of the slash (/) defined on H-spec.

**LastUpd** is a timestamp that is made up of **UpdDate** and **UpdTime**. Formats for the subfields *must* be \*ISO to match correctly the format for timestamps. (The microseconds portion of a Timestamp variable is padded with zeros during program initialization.)

**Reminder:** The \*ISO format is:

**yyyy-mmm-dd** for a Date

**hh.mm.ss** for a Time

**yyyy-mm-dd-hh.mm.ss.mmmmmmm** for a Timestamp

# Externally defined date and time fields

IBM i

- These fields are defined in DB using DDS or SQL.
- Formats of those fields can be defined externally also.
- When brought into an RPG program, format can be translated into default format of program:
  - For record formats with many dates, performance impact is possible
  - Similar behavior to numeric format translation
- Format translation can be avoided:
  - Ensure that default RPG format matches external formats whenever possible.
  - Use externally described DS to contain these fields:
    - External formats are retained in RPG for this case.
    - Redefine external date and time fields on a D spec with desired (external) format specified.

© Copyright IBM Corporation 2009

Figure 7-10. Externally defined date and time fields

AS075.0

## Notes:

When externally defined date or time fields are brought into an RPG program, their formats are normally translated from the external format (specified in DDS or SQL) to the RPG default format for this program if those formats are different. Especially for record formats with several date or time fields, this format translation on every I/O operation might negatively impact performance. In addition, the programmer might simply prefer to work with the externally defined format in the RPG program.

This format translation by RPG can be avoided, if desired, by using the methods described in this figure. Of course, if several different external formats are used in a given program, the option to match the H-spec format is not possible. In that case, the best solution in most cases is likely to use the externally described Data Structure, because it does not require the programmer hard-code the specific field names and formats. Subsequent changes to the external file description are more easily incorporated.

Examples illustrating this phenomenon follow.

Note that this behavior of translating the formats of date fields to the default for the program is the same kind of behavior that RPG has always done for numeric fields. Zoned numeric

fields from a database file are translated to packed under the same kind of circumstances as described here.

## Example: Externally defined dates (1 of 2)

IBM i

- DDS for file DateFlds
  - Contains two date fields: \*MDY and \*ISO formats
  - \*ISO is default format for RPG IV and DB2 (DDS)

```

H...+A*...1....+....2....+....3....+....4....+....5....+....6....+....7....+...
000100  * DDS FOR DATEFLDS
000200  A          R RECFMT
000300  A          DATEMDY      L          DATFMT(*MDY)
000400  A          DATEISO      L

```

- RPG IV translates formats for the date fields
  - Both fields display in \*EUR format (Default for program)

```

.....1....+....2....+....3....+....4....+....5....+....6....+....7....+...
000001  H DatFmt(*Eur)
000002  FDateFlds IF E           Disk
000003  /Free
000004  Read DateFlds;
000005  Dsply DateMDY '*REQUESTER'; // DateMDY = 29.04.2003
000006  Dsply DateISO '*REQUESTER'; // DateISO = 29.04.2003
000007  Eval *InLR = *On;
000008  /End-free
000009

```

© Copyright IBM Corporation 2009

Figure 7-11. Example: Externally defined dates (1 of 2)

AS075.0

### Notes:

This example illustrates what is presented in RPG IV when you refer to externally described date data type fields.

In this example, there is no D-spec definition or externally described data structure defined to hold the date fields. Therefore, the format of the two date fields in the file is changed to **\*EUR**, which is the default date format for this program, (as specified by the **DatFmt** keyword in the H-spec).

In the next example, the date formats are *not* translated when the date fields are copied in because we are using an externally described data structure.

## Example: Externally defined dates (2 of 2)

IBM i

- DDS for file DateFlds
  - Same two date fields as before

```

000100  * DDS FOR DATEFLDS
000200  A          RECFMT
000300  A          DATEMDY      L      DATFMT(*MDY)
000400  A          DATEISO      L

```

- RPG IV brings in external description for DateFlds
  - Fields display with externally described formats: \*MDY and \*ISO
  - Externally described data structure (Dates) is the difference

```

000001  H DatFmt(*Eur)
000002  FDateFlds IF E           disk
000003  DDates      E DS        ExtName(DateFlds)
000004  /Free
000005  Read DateFlds;
000006  Dsply DateMDY '*REQUESTER'; // DateMDY = 04/29/03
000007  Dsply DateISO  '*REQUESTER'; // DateISO = 2003-04-29
000008  *InLR = *On;
000009  /End-free

```

© Copyright IBM Corporation 2009

Figure 7-12. Example: Externally defined dates (2 of 2)

AS075.0

### Notes:

In this example, we avoid the translation of formats by bringing the record format into an externally described data structure.

RPG IV leaves formats in their external forms in data structures. The translation in the previous figure occurs only on I- and O-specs for externally described files. (Remember, I- and O-specs are generated for us by the RPG IV compiler.)

# Moving data to and from date, time, and timestamp

IBM i

- Like to like with format conversion:
  - Date to Date, Time to Time, Timestamp to Timestamp
  - Date or Time to Timestamp
  - Timestamp to Date or Time
- Date or Time to Character or Numeric
  - Use %Char, %SubDt, %Dec
- Character or numeric to Date, Time, or Timestamp:
  - Use %Date, %Time, %Timestamp
  - Can specify format of input
  - Value returned always in ISO format

© Copyright IBM Corporation 2009

Figure 7-13. Moving data to and from date, time, and timestamp

AS075.0

## Notes:

Built-in-Functions can help you to move date, time, and timestamp data in these data types to or from numeric and character fields. The system knows about proper date, time and timestamp data types but, it does not know how the date is formatted in a numeric field. Is it MDY, YMD, 4-digit year? For a character field, we must tell the system not only the format but also whether there is a separator character and what it is.

When moving data from a date to another date, for example, no format of the date needs to be specified as they are stored in a common format. In fact, the parameter (for a BIF usually), which would contain the format, must be blank if both the source and the target of the move are Date, Time, or Timestamp fields.

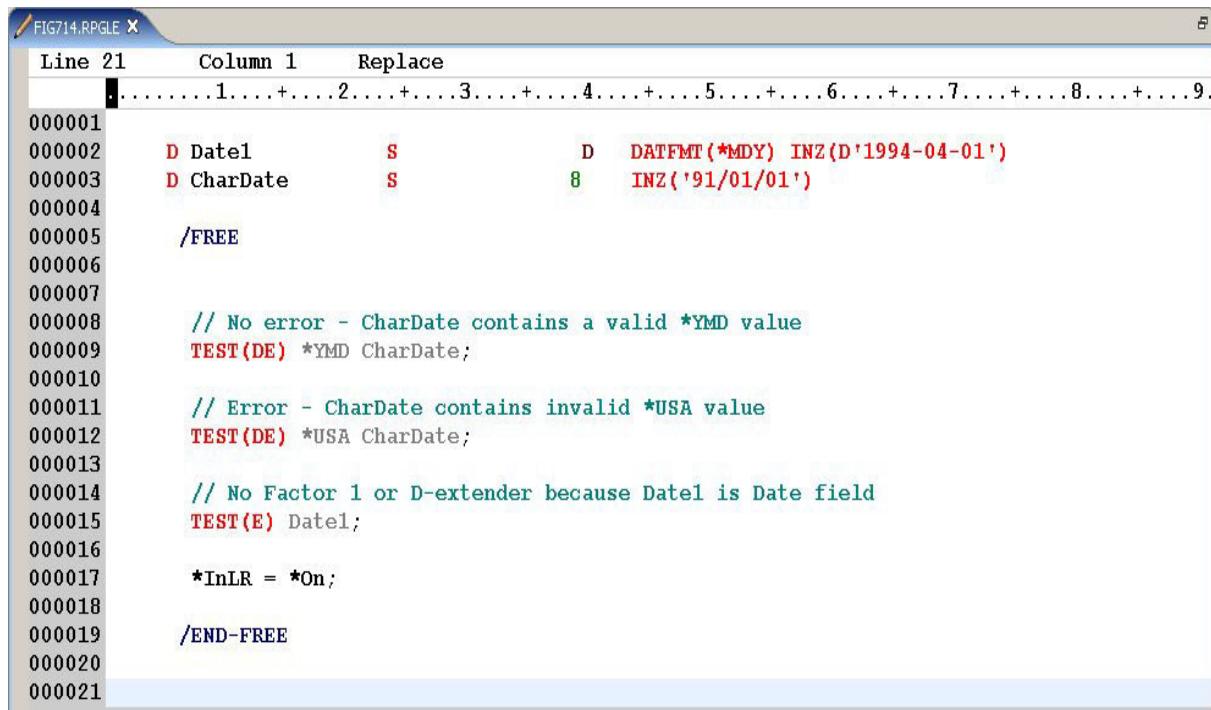
A 2-digit year format (\*MDY, \*DMY, \*YMD and \*JUL) can only represent dates 1940 to 2039. An error occurs if conversion from a 4-digit year to a 2-digit year format results in a year outside this range.

Because the system does not know the format of the date or time data in the numeric or alphanumeric field, you must explicitly declare the format that applies. If you do *not* declare

the format, the system assumes that the format matches that of the date or time field to which you are assigning the value.

# Testing for valid dates

IBM i



```

FIG714.RPGLE X
Line 21      Column 1      Replace
. ....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8....+....9.

000001
000002     D Date1          S           D   DATFMT(*MDY) INZ(D'1994-04-01')
000003     D CharDate        S           8   INZ('91/01/01')
000004
000005     /FREE
000006
000007
000008 // No error - CharDate contains a valid *YMD value
000009 TEST(DE) *YMD CharDate;
000010
000011 // Error - CharDate contains invalid *USA value
000012 TEST(DE) *USA CharDate;
000013
000014 // No Factor 1 or D-extender because Date1 is Date field
000015 TEST(E) Date1;
000016
000017 *InLR = *On;
000018
000019 /END-FREE
000020
000021

```

© Copyright IBM Corporation 2009

Figure 7-14. Testing for valid dates

AS075.0

## Notes:

You test for a zero divisor before you perform a division to avoid a divide by zero error.

You should also test fields that are moved to date, time, or timestamp fields before you perform the operation. To test for a valid date, time, or timestamp value use the TEST opcode with an extension of D,T or Z. The result field might be a date, time, timestamp, or a numeric field. You must use the E-extender and the %error BIF (%error not shown).

%Error is turned off if the date/time/timestamp is valid. Factor 1 defines the result field format.

Look at the line of code:

```
TEST(DE) *YMD CharDate;
```

What separator is being checked for in this code, / or &? The answer is / since this is the default separator for the \*YMD format. Any H-spec format does not apply.

Character or numeric fields can be tested to see if they contain a valid date, time or timestamp. The purpose of this would be to avoid getting an error on a move to a date or

time field, typically. If you are unsure of the format of the date in a character or numeric field, you could test for various formats.

Also, date, time, and timestamp fields can be tested for valid values.

# Setting date fields to job and system dates

IBM i

```

000001    D Date1      S          D  Inz(*$sys)
000002    D Date2      S          D  Inz(*Job)
000003    D Date3      S          D
000004    D Date4      S          D
000005    D Date5      S          D
000006
000007    /Free
000008
000009    Date3 = %Date(*Date);
000010    Date4 = %Date;
000011    Date5 = %Date(Udate);
000012    Dsply Date1 '*REQUESTER';
000013    Dsply Date2 '*REQUESTER';
000014    Dsply Date3 '*REQUESTER';
000015    Dsply Date4 '*REQUESTER';
000016    Dsply Date5 '*REQUESTER';
000017    // All dates have value 2003-04-29
000018    *InLR = *On;
000019    /End-free

```

© Copyright IBM Corporation 2009

Figure 7-15. Setting date fields to job and system dates

AS075.0

## Notes:

You can initialize the value of a date field using system values, **\*Sys** and **\*Job**, on the D-spec. You cannot use them in calculations with the **%date** BIF. However, the system date is assumed if no parameter is specified for the BIF, that is, **%date()**.

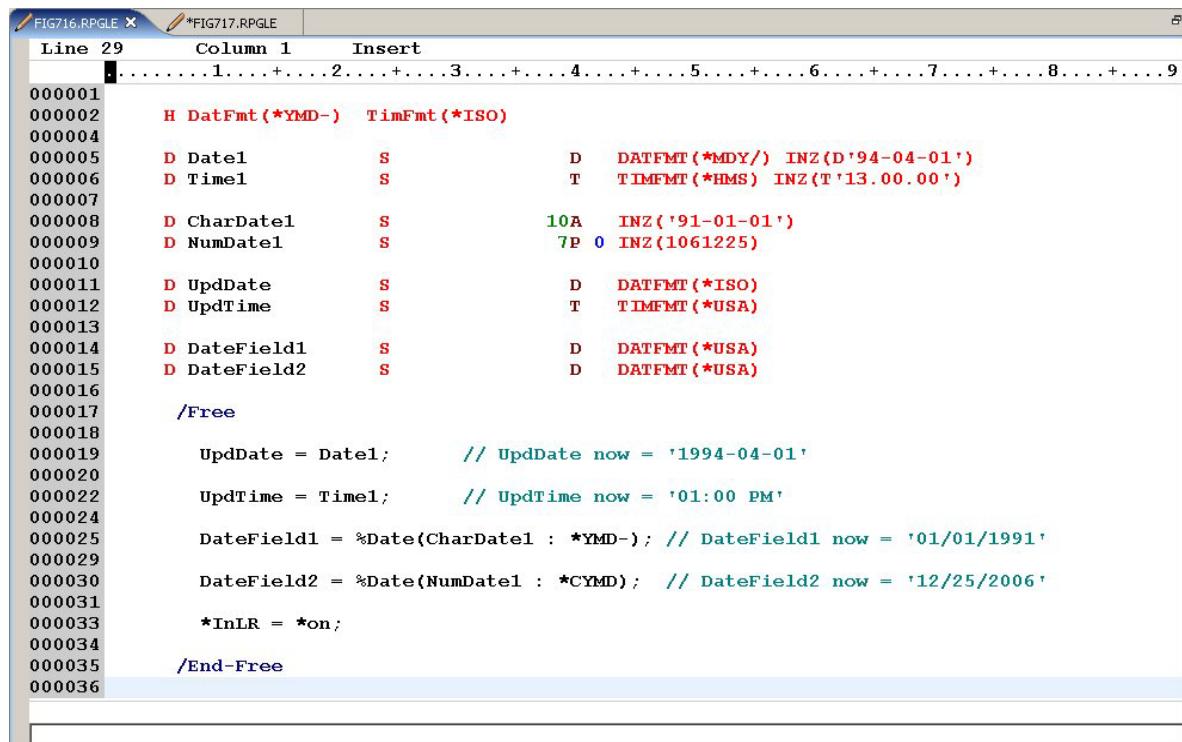
Note the use of **%date** with the **\*date** (8-digit numeric) and **Udate** (6-digit numeric) reserved words for the job date. **%date** converts numeric or character dates to date data types.

If the date is currently 04-29-2003, then all date fields in the example above will be set to 2003-04-29 (\*ISO format is the default). **%date** can accept an optional second parameter to specify a format for the numeric or character field on which **%date** operates.

**%date** always returns the date value in \*ISO format regardless of the format specified as the second parameter.

# Moving to and from date and time fields (1 of 2)

IBM i



```

FIG716.RPGLE *FIG717.RPGLE
Line 29 Column 1 Insert
000001 H DatFmt(*YMD-) TimFmt(*ISO)
000002 D Date1      S          D DATFMT(*MDY/) INZ(D'94-04-01')
000003 D Time1      S          T TIMEFMT(*HMS) INZ(T'13.00.00')
000004
000005 D CharDate1   S          10A INZ('91-01-01')
000006 D NumDate1    S          7P 0 INZ(1061225)
000007
000008 D UpdDate     S          D DATFMT(*ISO)
000009 D UpdTIme     S          T TIMEFMT(*USA)
000010
000011 D DateField1  S          D DATFMT(*USA)
000012 D DateField2  S          D DATFMT(*USA)
000013
000014 /Free
000015
000016
000017
000018 UpdDate = Date1; // UpdDate now = '1994-04-01'
000019 UpdTIme = Time1; // UpdTIme now = '01:00 PM'
000020
000021
000022
000023
000024
000025 DateField1 = %Date(CharDate1 : *YMD-); // DateField1 now = '01/01/1991'
000026
000027 DateField2 = %Date(NumDate1 : *CYMD); // DateField2 now = '12/25/2006'
000028
000029 *InLR = *on;
000030
000031
000032
000033
000034
000035
000036

```

© Copyright IBM Corporation 2009

Figure 7-16. Moving to and from date and time fields (1 of 2)

AS075.0

## Notes:

For **UpdDate** and **UpdTIme**, the assignment is *direct* as both **Date1** and **Time1** are date/timestamp variables.

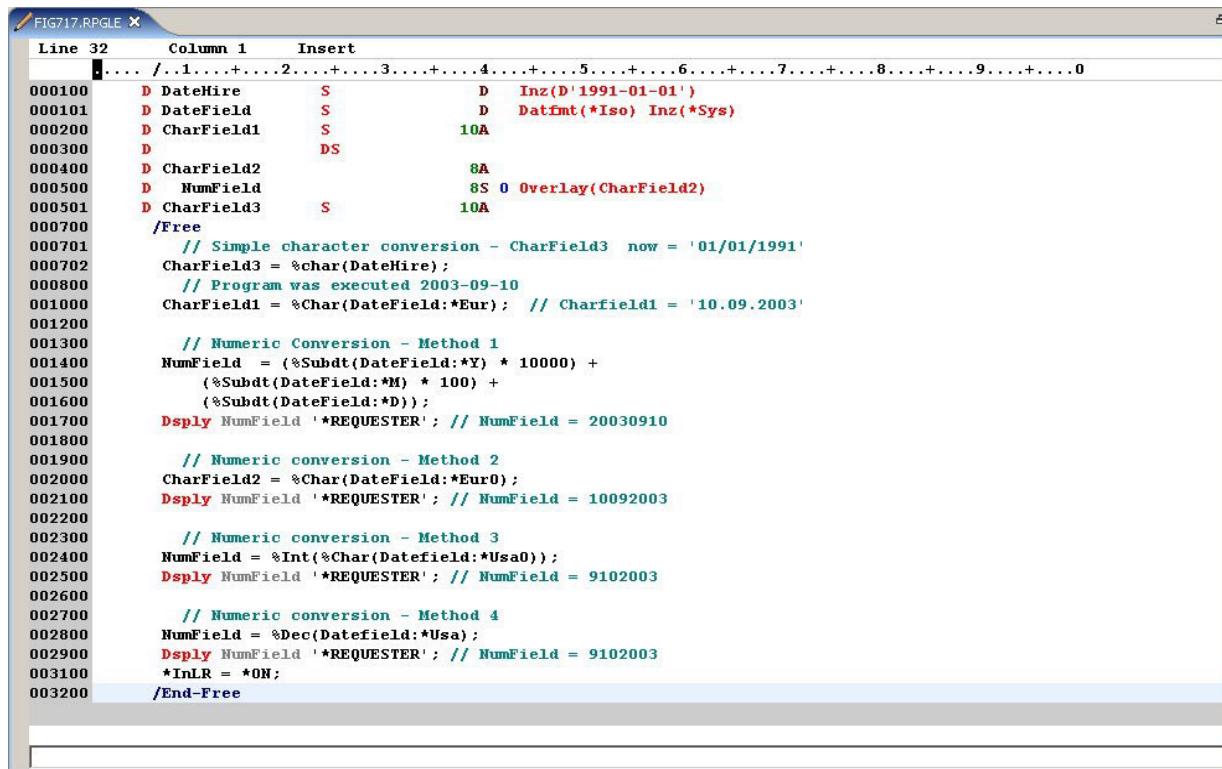
(It is never necessary to specify a format for a timestamp because there is only one format available.)

We know from the **INZ** value set in the D-spec that we have valid values. But, never forget the importance of testing the value **before** attempting the **Eval** operation. If the character field contained invalid date data (including blanks or zeroes), the assignment to the date data type fails.

In the other statements, BIFs are used to convert from one format to the other. The **%date** BIF converts a numeric or character date to a date data type variable. You should specify the date format of the numeric or character variable to be converted. If you do not specify a format, the default value is either the format specified on the H-spec **DATFMT** keyword or **\*ISO** (H-spec default).

## Moving to and from date and time fields (2 of 2)

IBM i



```

FIG717.RPGLE X
Line 32      Column 1      Insert
. .... /...1....+....2....+....3....+....4....+....5....+....6....+....7....+....8....+....9....+....0
000100      D DateHire      S          D Inz('1991-01-01')
000101      D DateField     S          D Datfmt(*Iso) Inz(*Sys)
000200      D CharField1    S          10A
000300      D                   DS
000400      D CharField2    8A
000500      D NumField       8S 0 Overlay(CharField2)
000501      D CharField3    S          10A
000700      /Free
000701      // Simple character conversion - CharField3 now = '01/01/1991'
000702      CharField3 = %char(DateHire);
000800      // Program was executed 2003-09-10
001000      CharField1 = %Char(DateField:*Eur); // Charfield1 = '10.09.2003'
001200
001300      // Numeric Conversion - Method 1
001400      NumField = (%Subdt(DateField:*Y) * 10000) +
001500      (%Subdt(DateField:*M) * 100) +
001600      (%Subdt(DateField:*D));
001700      Dsplay NumField '*REQUESTER'; // NumField = 20030910
001800
001900      // Numeric conversion - Method 2
002000      CharField2 = %Char(DateField:*Eur0);
002100      Dsplay NumField '*REQUESTER'; // NumField = 10092003
002200
002300      // Numeric conversion - Method 3
002400      NumField = %Int(%Char(Datefield:*Usa0));
002500      Dsplay NumField '*REQUESTER'; // NumField = 9102003
002600
002700      // Numeric conversion - Method 4
002800      NumField = %Dec(Datefield:*Usa);
002900      Dsplay NumField '*REQUESTER'; // NumField = 9102003
003100      *InLR = *ON;
003200      /End-Free

```

© Copyright IBM Corporation 2009

Figure 7-17. Moving to and from date and time fields (2 of 2)

AS075.0

### Notes:

Notice the use of **%Char** to move the value in a date field to a character field.

In this example, we focus on moving date data to character and numeric fields.

Moving to a character field can be easily accomplished using the **%Char** BIF, specifying an optional format as desired.

Moving to a numeric field is not quite as straightforward. There are several different methods shown. The first uses a formula to do the conversion. Notice the **%SubDT** BIF which extracts a portion (years, months, or days) from a date field. **%SubDT** returns a numeric value which can be used in the formula.

A better way to do the conversion to a numeric is to convert it to a character but to use the **OVERLAY** keyword to redefine the character value as a numeric on the D-spec.

The third way uses the **%Int** BIF in conjunction with **%Char**. Another form of the code that would convert today's date to an integer:

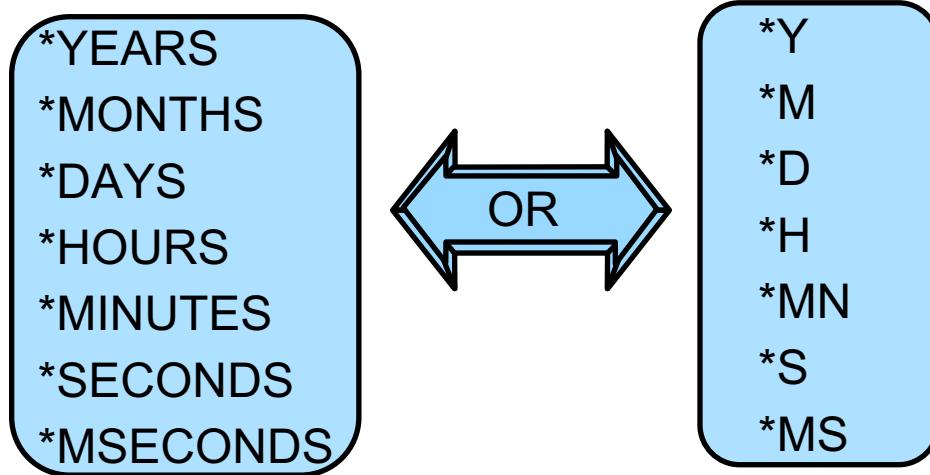
```
NumField = %int (%char (%Date() :*usa0));
```

Methods 3 and 4 illustrate using the **%Dec** and **%Int** BIFs to convert character data to numeric. Character expressions can now be a parameter of these BIFs.

The fourth example uses the **%Dec** BIF. **%Dec** was enhanced at V5R3 to accept a date, time or timestamp as the first parameter. The optional second parameter can be used to specify the format of the resulting numeric value.

## Date and time durations

- Possibilities:



- Specify as subfactor in factor 2 or result field
- Use as parameter with %Diff and %SubDt BIFs

© Copyright IBM Corporation 2009

Figure 7-18. Date and time durations

AS075.0

### Notes:

This visual summarizes how you specify the duration is to be expressed. We will cover some examples in the visuals that follow.

Durations can be determined or used in the **%Diff** BIF. They can be added to or subtracted from date, time, or timestamp fields. They can be the result of adding or subtracting from two date, time, or timestamp fields.

The **%SubDt** BIF is used to extract a portion of a date, time, or timestamp.

# Adding durations

IBM i

```

000001  D OrderDate      S          D  Inz(D'2001-07-31')
000002  D Shipdate       S          D
000003  D WarrantyDate   S          D
000004  D StartTime      S          T  Inz(T'12.40.00')
000005  D EndTime        S          T
000006  D MyTimeStamp    S          Z  Inz(*Sys)
000007  D NoYears         S          2S 0 Inz(2)
000008  D NoMinutes       S          2S 0 Inz(60)
000009
000010 /FREE
000011
000012     ShipDate = OrderDate + %days(7) + %months(1);
000013     WarrantyDate = ShipDate + %Years(NoYears);
000014     EndTime = StartTime + %Hours(8);
000015     MyTimeStamp = MyTimeStamp + %minutes(NoMinutes);
000016
000017     *InLR = *on;
000018 /END-FREE

```

© Copyright IBM Corporation 2009

Figure 7-19. Adding durations

AS075.0

## Notes:

Using expressions and BIFs, you can:

- Add days, months, or years to Date field
- Add hours, minutes, or seconds to Time field
- Add above and microseconds to Timestamp field

A duration addition might result in an invalid date. When this happens, the system automatically adjusts the resulting date. It is important that you understand how this works when you are writing your code. You might need to write additional code to handle a specific situation.

Here are some simple examples:

```

1996-02-29 plus 1 year = 1997-02-28
1996-01-30 plus 1 month = 1996-02-29

```

The field that holds the duration is defined as a normal numeric variable (for example, packed, zoned, or binary).

## Using %Days, %Years, %Months BIFs

While working with the date, time, and timestamp BIFs, you will discover that these lines of code will generate a compile error:

```
WarrantyDate = %Years (NoYears) + ShipDate  
EndTime = %Hours (8) + StartTime  
MyTimStamp = %minutes (NoMinutes) + MyTimStam
```

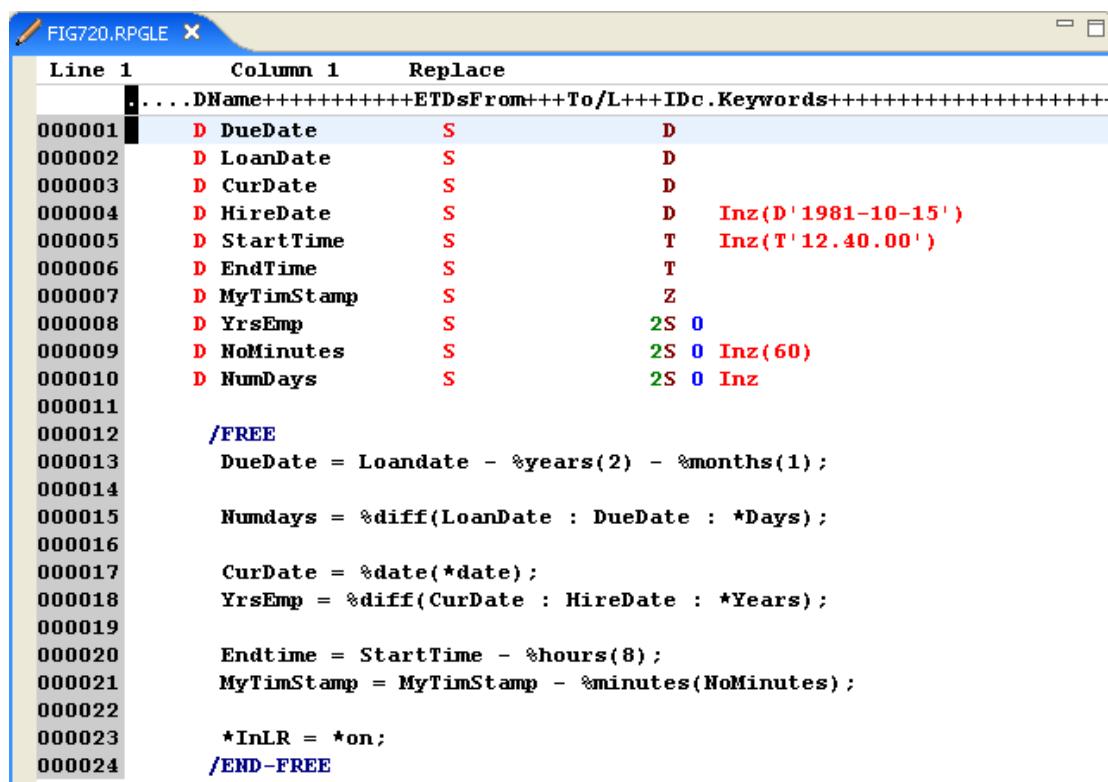
Each of the above lines of code generates this compile error:

```
RNF7421 30 Operands are not compatible with the type of operator.  
RNF7416 30 The types of the right and left hand side do not match in the  
EVAL operation.
```

As long as the BIF is not the first operand in the expression, the code compiles. This is the way these BIFs are designed to work and it is explained in the manual. For example, for %Months, the manual states that *%MONTHS can only be the right-hand value in an addition or subtraction operation. The left-hand value must be a date or timestamp.*

This rule is true for both addition and subtraction operations.

# Subtracting durations



```

FIG720.RPGL X
Line 1      Column 1      Replace
. ....DName++++++ETDsFrom+++To/L+++IDc.Keywords+++++
000001     D DueDate      S          D
000002     D LoanDate     S          D
000003     D CurDate      S          D
000004     D HireDate     S          D  Inz(D'1981-10-15')
000005     D StartTime    S          T  Inz(T'12.40.00')
000006     D EndTime      S          T
000007     D MyTimeStamp  S          Z
000008     D YrsEmp       S          2S 0
000009     D NoMinutes    S          2S 0 Inz(60)
000010     D NumDays      S          2S 0 Inz
000011
000012 /FREE
000013   DueDate = Loandate - %years(2) - %months(1);
000014
000015   Numdays = %diff(LoanDate : DueDate : *Days);
000016
000017   CurDate = %date(*date);
000018   YrsEmp = %diff(CurDate : HireDate : *Years);
000019
000020   Endtime = StartTime - %hours(8);
000021   MyTimeStamp = MyTimeStamp - %minutes(NoMinutes);
000022
000023   *InLR = *on;
000024 /END-FREE

```

© Copyright IBM Corporation 2009

Figure 7-20. Subtracting durations

AS075.0

## Notes:

You can:

- Subtract durations from Date, Time, and Timestamp
- Calculate durations between:
  - Two Dates, Times, and Timestamps
  - Date and Timestamp
  - Time and Timestamp

When calculating a duration between two dates or times, the result field *must* be a numeric field (or array or array element) with zero decimal positions.

The system automatically adjusts the date if the calculated date is invalid:

1995-03-29 minus 1 month = 1995-02-28  
 1995-02-28 minus 3 years = 1992-02-28

The **%diff** BIF determines the difference between two dates, times, or timestamps. The second parameter is subtracted from the first to determine the duration based on the third parameter. In the visual:

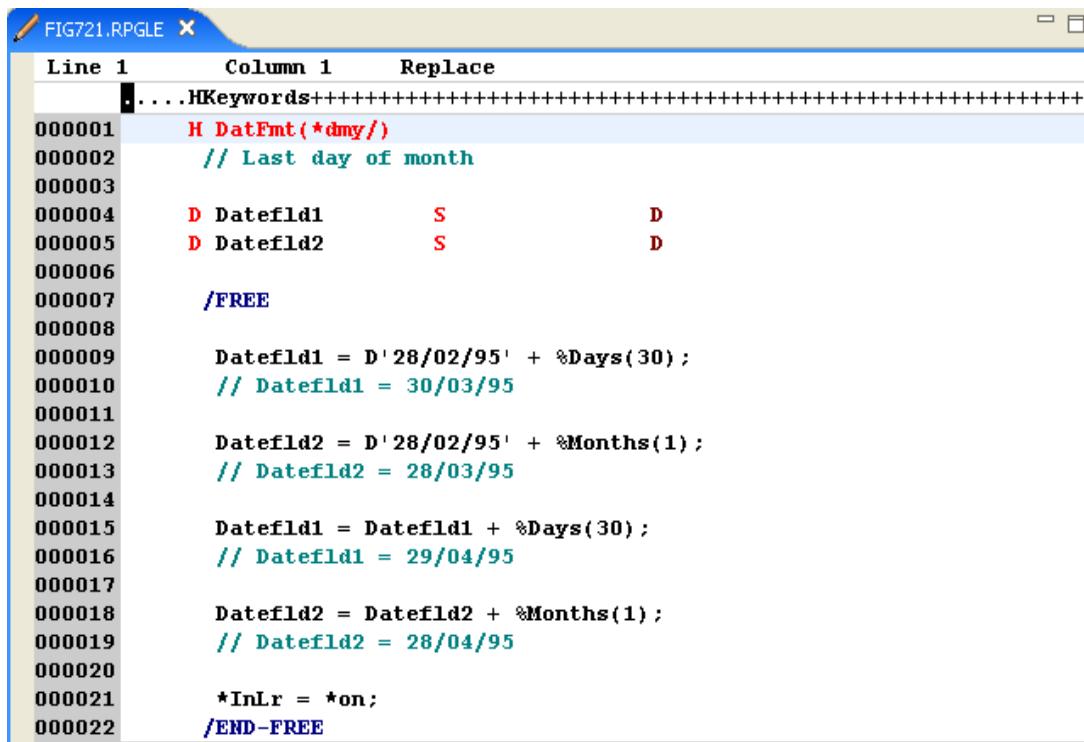
```
YrsEmp = %diff(CurDate : HireDate : *Years);
```

**YrsEmp** is expressed in years.

Note that special fields **\*DATE** and **UDATE** are numeric fields and are *not* date data types. Therefore, their values must be converted using **%date**.

# Duration considerations (1 of 2)

IBM i



```

FIG721.RPGLE X
Line 1      Column 1      Replace
. ....HKeywords+++++
000001      H DatFmt(*dmy/)
000002      // Last day of month
000003
000004      D Datefld1      S          D
000005      D Datefld2      S          D
000006
000007      /FREE
000008
000009      Datefld1 = D'28/02/95' + %Days(30);
000010      // Datefld1 = 30/03/95
000011
000012      Datefld2 = D'28/02/95' + %Months(1);
000013      // Datefld2 = 28/03/95
000014
000015      Datefld1 = Datefld1 + %Days(30);
000016      // Datefld1 = 29/04/95
000017
000018      Datefld2 = Datefld2 + %Months(1);
000019      // Datefld2 = 28/04/95
000020
000021      *InLr = *on;
000022      /END-FREE

```

© Copyright IBM Corporation 2009

Figure 7-21. Duration considerations (1 of 2)

AS075.0

## Notes:

When you calculate durations, the results might differ from your expectations.

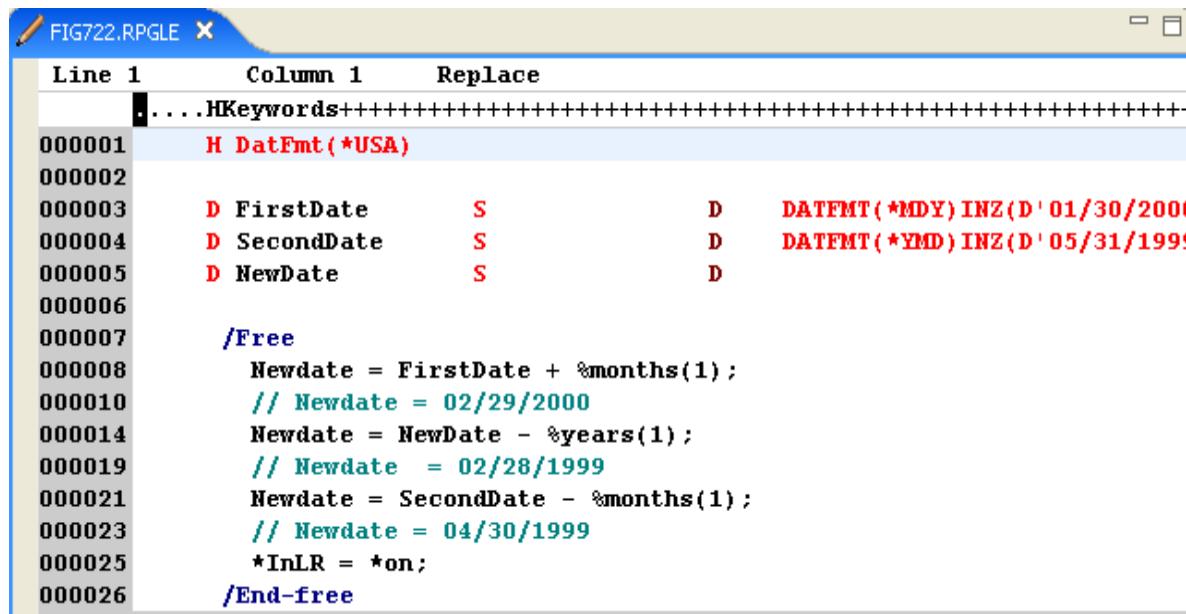
The above code illustrates the results of several duration calculations. We show another example next.

## References:

*ILE RPG Reference Manual, Chapter 20. Operations - Date Operations*

## Duration considerations (2 of 2)

IBM i



The screenshot shows an IBM i terminal window titled 'FIG722.RPGLE'. The code is as follows:

```

Line 1      Column 1      Replace
. ....HKeywords+++++
000001      H DatFmt(*USA)
000002
000003      D FirstDate      S          D      DATFMT(*MDY) INZ(D'01/30/2000)
000004      D SecondDate     S          D      DATFMT(*YMD) INZ(D'05/31/1999)
000005      D NewDate        S          D
000006
000007      /Free
000008      Newdate = FirstDate + %months(1);
000010      // Newdate = 02/29/2000
000014      Newdate = NewDate - %years(1);
000019      // Newdate = 02/28/1999
000021      Newdate = SecondDate - %months(1);
000023      // Newdate = 04/30/1999
000025      *InLR = *on;
000026      /End-free

```

© Copyright IBM Corporation 2009

Figure 7-22. Duration considerations (2 of 2)

AS075.0

### Notes:

Following the addition of one month to January 30, 2000, the resulting value in **NewDate** is 02/29/2000. This is because February 30, 2000, is an invalid date value. Because 2000 is a leap year, the day is set to the last day of February 2000, which is 29.

Following the subtraction of one year from **NewDate** (which contains 02/29/2000 from the step above) the value in **NewDate** becomes 02/28/1999. Similar to the case above, the subtraction of a year from 2000 without changing the resulting day results in an invalid date value. So the day value was set to the last valid day of February in 1999.

Following the subtraction of one month from May 31, 1999, the resulting value in **NewDate** will be 04/30/1999.

The results of duration calculations such as these may produce *interesting* results. However, the results of these situations are quite predictable and accurate. Doing similar duration calculations on date fields using the SQL language, for example, will produce similar results.

While the duration BIFs (**%Diff** and **%Years**, and so on) are easy to use and very powerful, there are some limitations that are described as *unexpected results* in the RPG Reference manual. The above code illustrates the unexpected results of a duration calculation.

## Durations based on today's date

IBM i

- UDATE and \*DATE are not date data types
  - 6-digit and 8-digit numeric values for Job Date
- Several choices:
  - Use INZ(\*SYS) or INZ(\*JOB) when defining the date field
  - Use TIME operation code to a date data type field
  - Use %DATE to assign UDATE or \*DATE to a date data type field

FIG723.RPGLE X

Line 1	Column 1	Replace
		. . . . DName++++++ETDsFrom+++To/L+++IDc .Keywords++++++
000100	D Today	S D DatFmt(*MDY) INZ(*Sys)
000101	D CurrentDate	S D DatFmt(*YMD)
000102	D TodaysDate	S D DatFmt(*USA) INZ(*JOB)
000103	D HireDate	S D DatFmt(*ISO) INZ(D'1982-02-28)
000104	D YrsEmp	S 2 0
000105		
000106	/Free	
000107	CurrentDate = %Date(*Date);	
000109	YrsEmp = %Diff(Today : HireDate : *Y);	
000111	YrsEmp = %Diff(CurrentDate : HireDate : *Y);	
000113	YrsEmp = %Diff(TodaysDate : HireDate : *Y);	
000116	*InLR = *On;	
000117	/End-free	

© Copyright IBM Corporation 2009

Figure 7-23. Durations based on today's date

AS075.0

### Notes:

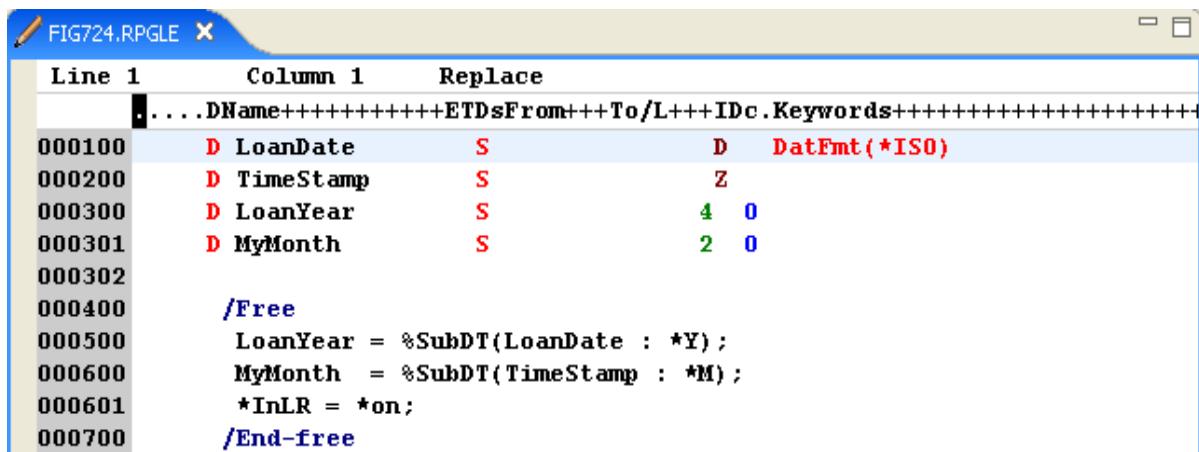
In the example, each of the three calculations for **YrsEmp** result in the same value, assuming that the job date and system date values are the same at the time the program is run. If you change the job date, then the value of **Today** and **TodaysDate** are impacted by that change.

Of course, there are other ways one could retrieve the system date, such as using an API or running a CL command. These three options are the most likely to be used within an RPG IV program.

# Extracting from dates, times, and timestamps

IBM i

- %SubDT BIF extracts subfield of Date, Time, or Timestamp
  - Year, month, or day of date or timestamp
  - Hour, minutes, or seconds of time or timestamp
  - Microseconds of timestamp



The screenshot shows an IBM i RPGLE editor window titled 'FIG724.RPGL'. The code is as follows:

```

Line 1      Column 1      Replace
. ....DName++++++ETDsFrom++To/L+++IDc .Keywords+++++
000100    D LoanDate      S           D   DatFmt(*ISO)
000200    DTimeStamp      S           Z
000300    D LoanYear       S           4  0
000301    D MyMonth        S           2  0
000302
000400    /Free
000500    LoanYear = %SubDT(LoanDate : *Y);
000600    MyMonth = %SubDT(TimeStamp : *M);
000601    *InLR = *on;
000700    /End-free

```

© Copyright IBM Corporation 2009

Figure 7-24. Extracting from dates, times, and timestamps

AS075.0

## Notes:

The **%SubDt** BIF places subfields of dates, times, or timestamps into the result field. The portion of the field desired is specified as the second parameter of the %SubDT BIF.

**%SubDT** is similar to a substring operation, but you do not have to specify a starting point or a length. In this case, the **LoanYear** field holds a 4-digit year. If it was defined as a 2-digit field in error, the compiler would issue a message.

# Comparing dates and times

IBM i

```

FIG725.RPGLE X
Line 1      Column 1      Replace
. ....HKeywords+++++.....HKeywords+++++.....HKeywords+++++
000100      H DatFmt(*MDY)
000101
000102      D Update      S          D  DatFmt(*USA)
000104      D TodaysDate   S          D  DatFmt(*MDY) INZ(*Sys)
000105      D ProcDate    S          D  Inz(*Job)
000106
000107      /Free
000108
000109      If Update < TodaysDate;
000110
000111      EndIf;
000112
000113      DoU ProcDate >= D'12/31/01';
000115
000116      EndDo;
000117
000118      *InLr = *on;
000119      /End-free

```

© Copyright IBM Corporation 2009

Figure 7-25. Comparing dates and times

AS075.0

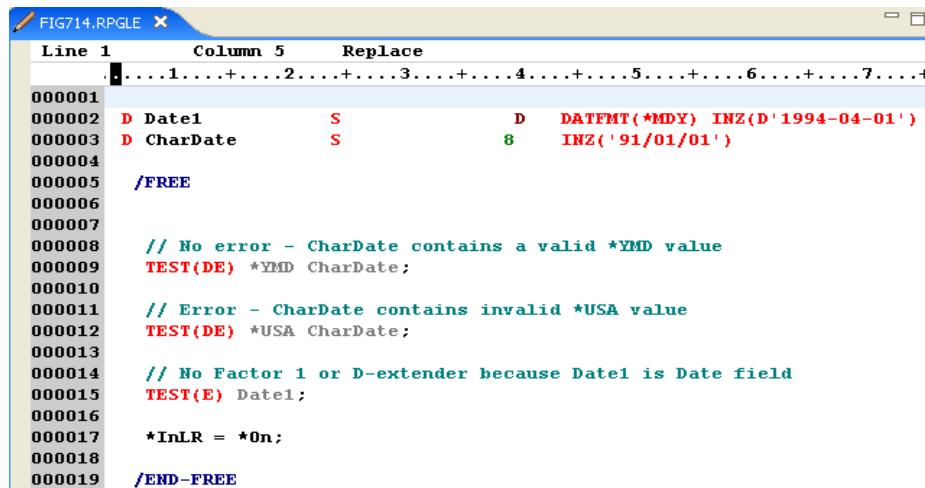
## Notes:

When comparing date or time data type fields, it is *not* necessary for both fields to be in the same format.

For example, the compiler can correctly compare a date in \*YMD format with a date in \*DMY format. The reason for this is that the dates are stored in a common format on the system. The date format is used only when the date is presented by the program to the user.

# Testing for valid values for date, time, and timestamp

- Use Test Opcode
  - D, T, or Z Extender
  - Factor1 specifies format (unless testing date, time, or timestamp data)
  - Result can be character, numeric or date, time, or timestamp
- Can add E-extender and use %Error



```

FIG714.RPGLE X
Line 1   Column 5   Replace
. ....1....+....2....+....3....+....4....+....5....+....6....+....7....+
000001
000002 D Date1      S           D  DATEFMT(*MDY) INZ(D'1994-04-01')
000003 D CharDate    S           8  INZ('91/01/01')
000004 /FREE
000005
000006
000007
000008 // No error - CharDate contains a valid *YMD value
TEST(DE) *YMD CharDate;
000009
000010
000011 // Error - CharDate contains invalid *USA value
TEST(DE) *USA CharDate;
000012
000013
000014 // No Factor 1 or D-extender because Date1 is Date field
TEST(E) Date1;
000015
000016
000017 *InLR = *On;
000018
000019 /END-FREE

```

© Copyright IBM Corporation 2009

Figure 7-26. Testing for valid values for date, time, and timestamp

AS075.0

## Notes:

We introduced **TEST** earlier. Let's review it again quickly.

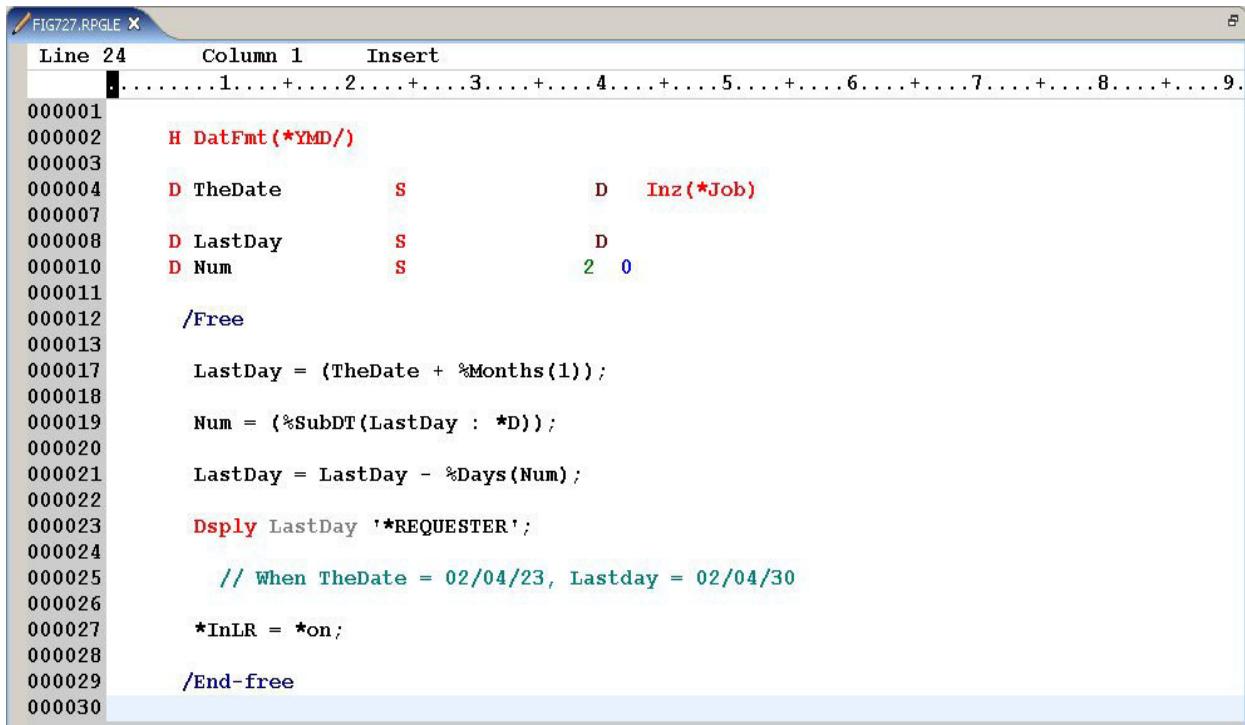
Notice that the **TEST(D)** operation should be used to test character or numeric type fields before assigning them. Any attempt to assign an invalid date value to a date field will result in a run-time error.

The Operation Code Extender (**E**) is a method of detecting error conditions.

The **TEST** operation code can also check time and timestamp values.

# Calculate last day of any month

IBM i



```

FIG727.RPGLE X
Line 24      Column 1      Insert
.....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8....+....9.

000001
000002      H DatFmt(*YMD)
000003
000004      D TheDate      S          D   Inz(*Job)
000007
000008      D LastDay      S          D
000010      D Num          S          2   0
000011
000012      /Free
000013
000017      LastDay = (TheDate + %Months(1));
000018
000019      Num = (%SubDT(LastDay : *D));
000020
000021      LastDay = LastDay - %Days(Num);
000022
000023      Dsply LastDay '*REQUESTER';
000024
000025      // When TheDate = 02/04/23, Lastday = 02/04/30
000026
000027      *InLR = *on;
000028
000029      /End-free
000030

```

© Copyright IBM Corporation 2009

Figure 7-27. Calculate last day of any month

AS075.0

## Notes:

The program in this visual illustrates how to determine the last day of a given month (in this example, today's date). It further illustrates the use of duration expressions, %SubDt, and %days.

For many institutions, the end of a month means just that — the last calendar day of the month.

## Calculate day of week

IBM i

- Calculate day of week for any date
- Assume Sunday = day 7

```

FIG728.RPGLE X
Line 1      Column 1      Replace
. . . . HKeywords+++++Replace+++++Replace+++++
000001      H DatFmt( *MDY)
000002
000004      D AnySunday      S          D   Inz(D'04/21/02')
000005      D TodaysDate     S          D   Inz(*Job)
000006      D WorkNum        S          4   0
000007      D WorkDay        S          1   0
000008
000009      /Free
000010      WorkNum = %Diff(TodaysDate : AnySunday : *D);
000011      WorkDay = %Rem(WorkNum : 7);
000012      If WorkDay < 1;
000013          WorkDay = WorkDay +7;
000014      EndIf;
000015
000016      Dsply WorkNum 'REQUESTER';
000017      Dsply WorkDay 'REQUESTER';
000018      // If TodaysDate = 04/23/02, WorkDay = 2
000019      *InLR = *on;
000020      /End-free

```

© Copyright IBM Corporation 2009

Figure 7-28. Calculate day of week

AS075.0

### Notes:

This program calculates the day of week (Monday = 1, Tuesday = 2, and so forth) from any date. It takes a field, **TodaysDate** (a date data type field), as input and produces a resulting value of 1 through 7 to indicate the day of the week. Note that the value in the INZ keyword for **AnySunday** can contain any date value that is a Sunday.

If you want to change this code to represent a different sequence of days (for example, if you want Sunday to be day 1 rather than day 7), you must simply change the INZ value of the field **AnySunday** to be any date for the day you wish to be day 7.

Of course, you should also change the name of that field, as it would seem ridiculous to have a field called **AnySunday** that contained a date value for a Saturday.

# Calculate age

IBM i

```

100100  FAgeInq  CF   E          Workstn  IndDS(WKIIND)
100200
100300  D WkInd      DS
100400  D Exit       3      3N
100500  D BadDate    40    40N
100600  D BornMonth  S      2  0
100700  D CurrMonth  S      2  0
100800
100900  /Free
101000  Write   Header;
101100  Write   Footer;
101200  ExFmt   Prompt;
101300
101400  Dow NOT Exit;
101500  // Test for valid date value
101600>>1  Test(DE) *ISO Born;
101700  If %error;
101800>>2  BadDate = *On;
101900  Else;
102000  // Display details
102100>>3  Age = %Diff(%date(*date) : %date(Born : *ISO):*y);
102200  Months = %Diff(%date(*date) : %date(Born:*ISO):*m)
102300  - Age*12;
102400  Write Detail;
102500  Endif;
102600  // Display prompt
102700  ExFmt Prompt;
102800  Enddo;
102900
103000  *InLR = *On;
103100  Return;
103200  /End-Free

```

© Copyright IBM Corporation 2009

Figure 7-29. Calculate age

AS075.0

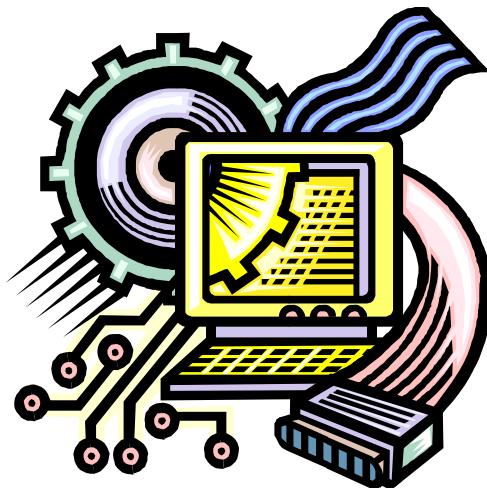
## Notes:

The program uses many new date handling facilities. Notice:

1. We use the TEST opcode with the DE extender to check for a valid date. The date field is entered as a character variable on the display. Notice the use of the format in the test opcode to ensure that the date that is entered is in \*ISO format.
2. If the value entered is not valid, we set on an indicator, **BadDate**, and the DSPF issues a message.
3. If the value entered is a valid date, we convert it to a date using the **%date** BIF. We also use **\*date** to get the job date and convert it using **%date**. Notice that we are calculating the values for **Age** in years and for **Months** in months. **%date** can also convert from a numeric data type to a date data type.

# Machine exercise: Using dates

IBM i



© Copyright IBM Corporation 2009

Figure 7-30. Machine exercise: Using dates

AS075.0

## Notes:

Perform the machine exercise “Using dates.”

## Unit summary

Having completed this unit, you should be able to:

- Code definitions for date, time, and timestamp data types
- Explain the difference between numeric or character date or time fields and the data types DATE and TIME
- Use the %Diff BIF to calculate date and time durations
- Use %DATE, %TIME, %TIMESTAMP to convert characters to date, time, and timestamp data types
- Use the %SubDt BIF to extract a subfield from a date, time, and timestamp data type
- Use %YEARS, %MONTHS, %DAYS, %HOURS, %MINUTES, %SECONDS, %MSECONDS to convert a numeric value into a duration
- Explain the date rounding process

© Copyright IBM Corporation 2009

Figure 7-31. Unit summary

AS075.0

### Notes:

Here are some good examples of date related conversions that can be found in the archives of the [www.midrange.com rpg-400I](http://www.midrange.com/rpg-400I). The subject of manipulation of fields containing date data and date data type fields is a subject of frequent discussion.

```
D #TmStamp      S          Z
D #TodayDate    S          D  DatFmt (*ISO)
D #TodayTime    S          T  TimFmt (*HMS)
D #DateEUR      S          D  DatFmt (*EUR)
D #DateISO      S          D  DatFmt (*ISO)
D #TimeISO      S          T  TimFmt (*ISO)

D #DatNumEUR    S          8S 0 Inz (22032001)
D #DatNumISO    S          8S 0
D #DatNum6      S          6S 0
D #DatNum7      S          7S 0
D #TimNum       S          6S 0 Inz (113614)

D #CharDate     S          8A   Inz (20020322') '
D #CharTime     S          6A   Inz (113614') '
```

```

D #CharDate10      S          10A
D FldChar6         S          6A
D FldChar7         S          7A
D FldChar8         S          8A
D FldChar9         S          9A
D FldChar10        S          10A
D FldChar14        S          14A
D FldNum3          S          3S 0
D FldNum4          S          4S 0
D FldNum6          S          6S 0
D FldNum7          S          7S 0
D FldNum8          S          8S 0
D FldPack20        S          20P 0

D #CurCY           S          4S 0
D #CurMon          S          2S 0
D #CurDay           S          2S 0
D #Hours            S          10S 0
D #Minutes          S          2S 0
D #Seconds          S          2S 0
D #NumYears          S          11S 0
D #NumMonth          S          11S 0
D #NumDays           S          11S 0
D #NumHours          S          11S 0
D #NumMinutes         S          11S 0
D #NumSeconds         S          11S 0

D                   DS
D $StrTmStamp       Z          Inz
D #StrDate          D          Overlay($StrTmStamp:1)
D #StrTime          T          Overlay($StrTmStamp:12)
D                   DS
D $EndTmStamp        Z          Inz
D #EndDate          D          Overlay($EndTmStamp:1)
D #EndTime           T          Overlay($EndTmStamp:12)

D #Diff              10I 0

D $TimDiff           DS
D #DiffHours          7S 0
D #DiffMinutes         2S 0
D #DiffSeconds         2S 0

*-----
/free

// Retrieve current date
#TodayDate=%date();

// Convert numeric field into date field

```

```
#DatNumISO=20020325;
#DateISO=%date(#DatNumISO:*iso);

// Convert character field into date field
#CharDate=20020325';
#DateISO=%date(#CharDate:*iso0);

// Convert character field with date into numeric field with date
FldChar6=080303';
#DatNumISO=%int(%char(%date(FldChar6:*dmy0):*iso0));

// Convert numeric field with date into numeric field with date
#DatNumISO=20030315;
#DatNum6=%int(%char(%date(#DatNumISO:*iso):*dmy0));
#DatNum6=%int(%char(%date(#DatNumISO:*iso):*ymd0));

#DatNum7=1030704;
#DatNum6=%int(%char(%date(#DatNum7:*cymd):*dmy0));

// Convert character field to date field
#CharDate10=2002-03-25';
#DateISO=%date(#CharDate10:*iso);

// Convert current date to numeric field
#DatNumISO=%int(%char(%date():*iso0));

// Convert date field to numeric field
#DatNumISO=%int(%char(#DateISO:*iso0));
#DatNum6=%int(%char(#DateISO:*dmy0));
#DatNum7=%int(%char(#DateISO:*cymd0));

// Convert date field to character field with separator .
FldChar8=%char(#DateISO:*dmy.);

// Retrieve day of year into numeric field (from given date)
#DateISO=%date(20030725':*iso0)';
FldNum3=%int(%subst(%char(#DateISO:*jul0):3:3));

// Retrieve day of year into numeric field (from current date)
FldNum3=%int(%subst(%char(%date():*jul0):3:3));

// Retrieve current time
#TodayTime=%time();

// Convert current time numeric field
#TimNum=%int(%char(%time():*iso0));

// Convert current time numeric field (only HHMM)
FldNum4 = %int(%subst(%char(%time():*iso0):1:4));
// Convert numeric field into time field
#TodayTime=%time(#TimNum:*iso);
```

```

// Convert character field into time field
#TodayTime=%time(#CharTime:*iso0);

// Create individual timestamp. Example: DDMMHHMMSS
FldChar10 = %subst(%triml(%editw(%subdt(%date():*D):0')):'8:2) +
%subst(%triml(%editw(%subdt(%date():*M):0')):'8:2) +
%subst(%triml(%editw(%int(%char(%time()):*iso0)):'0')):4:6);'

// Add 3 days to #DatNum7 and convert the result to #DatNumISO
#DatNumISO = %int(%char(%date(#DatNum7:*cymd) + %days(3):*iso0));

// Add 3 days to current date and convert the result to a character field
FldChar8 = %char(%date() + %days(3):*iso0);

// Add seconds to a numeric field
#TimNum = %int(%char(%time(#TimNum:*iso) + %seconds(1):*iso0));

// Retrieve current timestamp
#TmStamp=%timestamp();

// Retrieve string YYYYMMDDHHMMSS from current timestamp
FldChar14 = %subst(%char(%timestamp():*iso0):1:14);

// Retrieve full timestamp into numeric field (Resultfield must be packed)

FldPack20 = %dec(%char(%timestamp()):*iso0):20:0;

// Date/Time codes: *YEARS (*Y), *MONTHS (*M), *DAYS (*D), *HOURS (*H),
//                   *MINUTES (*MN), *SECONDS (*S), *MSECONDS (*MS)

// Extract year/month/day from date field
#CurCY=%subdt(#TodayDate:*Y);
#CurMon=%subdt(#TodayDate:*M);
#CurDay=%subdt(#TodayDate:*D);

// Extract hours/minutes/seconds from time field
#Hours =%subdt(#TodayTime:*H);
#Minutes=%subdt(#TodayTime:*MN);
#Seconds=%subdt(#TodayTime:*S);

// Add/subtract years, months, days to date field
#DateISO=#DateISO+%years(1);
#DateISO=#DateISO+%months(6);
#DateISO=#DateISO+%days(30);

#DateISO=#DateISO-%years(1);
#DateISO=#DateISO-%months(6);

```

```
#DateISO=#DateISO-%days(30);

// Add/subtract hours, minutes, seconds
#TimeISO=#TimeISO+%hours(1);
#TimeISO=#TimeISO+%minutes(15);
#TimeISO=#TimeISO+%seconds(10);

#TimeISO=#TimeISO-%hours(1);
#TimeISO=#TimeISO-%minutes(15);
#TimeISO=#TimeISO-%seconds(10);

// Calculate date and time differences
#NumYears=%diff(#TodayDate:#DateISO:*Y);
#NumMonth=%diff(#TodayDate:#DateISO:*M);
#NumDays=%diff(#TodayDate:#DateISO:*D);

#NumHours=%diff(#TodayTime:#TimeISO:*H);
#NumMinutes=%diff(#TodayTime:#TimeISO:*MN)
#NumSeconds=%diff(#TodayTime:#TimeISO:*S);

// Test valid date in numeric fields
FldNum8=20020325;
test(de) *iso FldNum8;
if %error;
// ...
endif;

FldNum7=1020325;
test(de) *cymd FldNum7;
if %error;
// ...
endif;

FldNum6=250302;
test(de) *dmy FldNum6;
if %error;
// ...
endif;

// Test valid date in character fields
FldChar10='2002-03-25';
test(de) *iso FldChar10;
if %error;
// ...
endif;

FldChar8='20020325';
test(de) *iso0 FldChar8;
if %error;
// ...
endif;
```

```

FldChar9=102/03/25';
test(de) *cymd FldChar9;
if %error;
// . . .
endif;

FldChar7=1020325';
test(de) *cymd0 FldChar7;
if %error;
// . . .
endif;

FldChar8=25/03/02';
test(de) *dmy FldChar8;
if %error;
// . . .
endif;

FldChar6=250302';
test(de) *dmy0 FldChar6;
if %error;
// . . .
endif;

// Test valid time in numeric fields
FldNum6=103521;
test(te) *iso FldNum6;
if %error;
// . . .
endif;

// Test valid time in character fields
FldChar6=103521';
test(te) *iso0 FldChar6;
if %error;
// . . .
endif;

// Calculate time difference between start- and end date/time.
// Result fields are contained in DS $TimDiff
#EndDate=%date(20031011:*iso);
#EndTime=%time(060000:*iso);
#StrDate=%date(20031008:*iso);
#StrTime=%time(000000:*iso);

#Diff = %diff($EndTmStmp:$StrTmStmp:*seconds);
#DiffHours = #Diff/3600;
#DiffMinutes = (#Diff - #DiffHours * 3600) / 60;
#DiffSeconds = #Diff - #DiffHours * 3600 - #DiffMinutes * 60;

```

```
// Retrieve date of last day of a month
#TodayDate = %date(); // Example
#DateISO = (#TodayDate + %months(1)) -
    %days(%subdt(#TodayDate + %months(1) : *D));
Return;
/end-free
```

# Unit 8. Using prototyped program calls

## What this unit is about

This unit teaches you how to use RPG IV's prototyping support as a way to perform calls to other programs.

## What you should be able to do

After completing this unit, you should be able to:

- Create a prototype for calling a program
- Create a procedure interface for a program
- Explain the advantages of using prototypes

## How you will check your progress

- Machine exercise

# Unit objectives



IBM i

After completing this unit, you should be able to:

- Create a prototype for calling a program
- Create a procedure interface for a program
- Explain the advantages of using prototypes

© Copyright IBM Corporation 2009

---

Figure 8-1. Unit objectives

AS075.0

## Notes:

# Calls in RPG IV

IBM i

- Why call other programs?
  - Execute logic not included in your program
  - Write once, use many times
- How?
  - Dynamic CALL
    - CALL opcode: Legacy method
    - CALLP opcode: Uses prototyping to reduce errors
- Static CALL:
  - 'Bound Call'
  - Less overhead at run-time
  - Use CALLP or CALLB opcodes
  - Subprocedure calls

© Copyright IBM Corporation 2009

Figure 8-2. Calls in RPG IV

AS075.0

## Notes:

RPG IV gives you three different opcodes to call other programs (or ILE Modules). You have already seen the dynamic **CALL** in the subfile unit.

We discuss the prototyped **CALLP** opcode and prototyping in this unit.

# Calling another program

IBM i

- Calling program (caller):
  - Issues call to another program
  - Passes parameters
  - Parameters defined as prototype (PR)
- Called program (callee):
  - Receives parameters
  - Parameters defined as a procedure interface (PI)
- Parameter passing methods:
  - By reference (address) is most common
  - By read-only reference
  - By value

© Copyright IBM Corporation 2009

Figure 8-3. Calling another program

AS075.0

## Notes:

Parameters must be defined in both the program that makes the call as well as the program that is called. The normal way of passing parameters is accomplished by passing the addresses of the parameters. Both the called and the calling program can access the same address for an individual parameter.

# The calling program

IBM i

```

FIG85.RPGLE X
Line 1      Column 1      Replace
.....1....+....2....+....3....+....4....+....5....+....6....+....7...
000100
000200 >>1 D FindCust      PR                               ExtPgm('FINDC')
000300   D
000400   D
000500   D
000600
000700 >>2 D CustNo       S      8S 0
000800   D StopFlag       S      5S 0
000900   D RecExists     S      1N
001000
001100   D OtherField    S      40
001200
001300   /Free
001400 >>3 CallP FindCust (StopFlag : CustNo : RecExists);
001500   *InLR = *on;
001600   /End-free

```

© Copyright IBM Corporation 2009

Figure 8-4. The calling program

AS075.0

## Notes:

1. The calling program contains an exact copy of the **PR** from the called program. We see the called program next and you notice how the parameters are the same.
  2. Fields passed as parameters from the program must be defined. They can be defined in any order. What is critical is:
    - a. The order of the parameters specified in the CALLP
    - b. The definition of the parameters (data type and size), which matches the **PR/PI** in the called program
  3. Notice in the call, the CALLP (call prototype) opcode is used. You must call a program (the name of the PR). Notice that the name of the PR might be different than the actual name of the program (EXTPGM keyword). This is handy when you call system routines or functions that sometimes are not named the way you might want them to be named.
- When you compile the program, the compiler validates the parameter list (they are in parentheses separated by colons). The parameters must match in order and data type

or length to the list in the PPrototype. If you enter one or more parameters incorrectly, you will get a compile time error:

RNF7535E The type and attributes of parameter n do not match those of the prototype.

This feature alone makes prototyping an excellent way of performing calls.

Further, in order to avoid run time errors with a **CALLP** such as the called program is not found, you can use the **E-extender** and **%error** with the CALLP opcode.

**More information:** You can use the LIKE keyword to define parameters in the Prototype as well. Here is an example:

```
D DeleteProgram    PR          ExtPgm(DELETEPGM') '
D   Parm1           Like(VndNbr)
```

You must code a variable name for the parameter if you want to do this. Without the variable name, in this case, PARM1 (which does nothing), the compiler thinks LIKE(VNDNBR) is another keyword for the previous line of code and diagnoses an error.

# The called program

IBM i

```

FINDC.RPGLE X
Line 28      Column 1      Replace
. .... /...1....+....2....+....3....+....4....+....5....+....6....+....7....+....8....+....9.

000002
000003 //***** Program: FINDC *****
000004
000009      FCustFil  IF   E          K Disk
000010
000011 >>1 D FindCust      PR          ExtPgm( 'FINDC' )
000012 D
000013 D
000014 D
000015
000016 >>2 D FindCust      PI
000017 >>3 D EndFlag
000018 D CustNumber
000019 D RecFound
000020
000021 >>4 D Message       S          40     Inz( 'Program FINDC ended' )
000022
000023
000024 /Free
000025
000026     If EndFlag = 0;
000027         Chain CustNumber CustFil;
000028         RecFound = %Found(CustFil);
000029     Endif;
000030
000031     Dsply Message *REQUESTER';
000032     *InLr = *On;
000033
000034 /End-Free
000035

```

© Copyright IBM Corporation 2009

Figure 8-5. The called program

AS075.0

## Notes:

When you write a program that is called by other programs, you write it to be able to receive and return parameters by coding:

1. **Prototype:** The prototype is a description of how the parameters are passed by the programs that call this program. The prototype is defined exactly like the Procedure Interface. These PRs are copied to the calling program.

Notice that although the names of the variables might be different than those in the PR, the parameters match in data type, length, and number of decimal places.

2. **Procedure Interface:** The procedure interface is that part of the called program that defines how other programs must interface with this program. It describes exactly the data type, length, and other characteristics of each variable *required* by this program. Programs are known as procedures in ILE. We discuss this more in a later unit.



**Note**

The PI and PR must have the same name, to denote the definition of the parameters being input to the program. Very often, the PR and PI names would also be the same as the program name — FINDC in this example.

This program requires three parameters in the precise order that they are listed. The called program expects a 5-digit number with zero decimals, followed by a 8-digit number with zero decimals followed by an indicator variable.

The address of these variables will be supplied to the called program; that is, the calling program defines these variables, the compiler assigns a storage address for each one, and, at the time of the call, the calling program supplies the addresses of the three parameters above.

3. The variables in the PI are defined to the called program.
4. A standalone local variable MESSAGE is defined in the called program.

# Calling program calls called program

IBM i

```

>>1 D FindCust      PR          ExtPgm('FINDC')
D                                         5S 0
D                                         8S 0
D                                         1N
D

>>2 D CustNo        S           8S 0
D StopFlag       S           5S 0
D RecExists       S           1N

D OtherField      S           40

/FREE
CALLP FindCust (StopFlag : CustNo : RecExists);
*InLR = *On;
/End-Free

```

\*\*\*\*\* Program: FINDC \*\*\*\*\*

FCustFil	IF	E	K Disk
D FindCust	PR	ExtPgm('FINDC')	
D		5S 0	
D		8S 0	
D		1N	
D FindCust	PI		
D EndFlag		5S 0	
D CustNumber		8S 0	
D RecFound		1N	
D Message	S	40	Inz('Program FINDC ended')

```

/FREE
If EndFlag = 0;
  Chain CustNumber CustFil;
  RecFound = %Found(CustFil);
Endif;

Dsplay Message! *REQUESTER!;
*InLr = *On;
/End-Free

```

© Copyright IBM Corporation 2009

Figure 8-6. Calling program calls called program

AS075.0

## Notes:

In this visual, we illustrate the calling program calling the called program. Notice that each program can access the storage address of each of the three parameters. Both programs can modify the contents of any of the parameters.

In this case, the called program Chains to the CUSTFIL using the key supplied by the calling program. The Chain is successful and the value of the indicator is returned as '1'.

## Performance tip

If a variable that is a parameter is operated on frequently in the called program, you should create a second *local* variable in the program. When you are ready to return to the calling program, move the data from the local variable to the parameter variable.

# Return to calling program

IBM i

```

FIG87V2.RPGLE X
Line 28    Column 3      Insert
...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
000003 //***** Program: FINDC *****
000004
000009 FCustFil  IF   E          K Disk
000010
000011 D FindCust      PR          ExtPgm( 'FINDC' )
000012 D
000013 D
000014 D
000015
000016 D FindCust      PI
000017 D EndFlag        5S 0
000018 D CustNumber     8S 0
000019 D RecFound       1N
000021
000022 D Message        S          40      Inz( 'Program FINDC ended' )
000023
000024 /Free
000025   If EndFlag = 0;
000026     Chain CustNumber CustFil;
000027     RecFound = %Found(CustFil);
000028     *InLr = *Off;
000029
000030
000031 Else;
000032   Dsply Message *REQUESTER*;
000033   *InLr = *On;
000034 Endif;
000035
000036
000037 >> Return;
000038 /End-Free

```

© Copyright IBM Corporation 2009

Figure 8-7. Return to calling program

AS075.0

## Notes:

Because you can call a program repeatedly, you might not want to set on LR after the called program has completed. There is a way to leave the called program active.

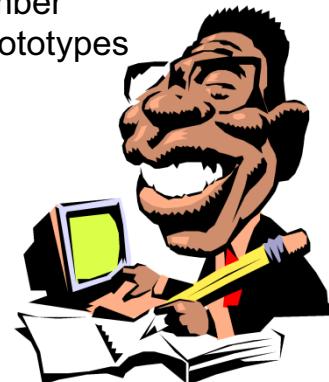
When you return to the caller without having turned on LR, the called program remains active. This is good if you intend to call the program repeatedly. You do not pay the price of initializing the called program more than once. This improves call performance.

The **Return** opcode returns control to the caller. In the example above, we can return to the caller without having set LR on. We only set LR on when the StopFlag is not equal to zero. Now you can see the use of this parameter.

# Advantages of prototyping

IBM i

- Parameters are validated at compile time; run-time problems are minimized.
- Prototypes tell the compiler to validate parameters:
  - Number of parameters
  - Data type and size
- Prototypes are easily held in separate members.
  - Use /COPY to bring into programs:
    - Can even store all prototypes in one copy member
    - Compiler does not care about unreferenced prototypes



© Copyright IBM Corporation 2009

Figure 8-8. Advantages of prototyping

AS075.0

## Notes:

As we have stated, prototyping is designed to help you to avoid run-time errors when passing parameters. The compiler validates that the prototyped parameters match those passed in the CALLP opcode.

Many RPG IV programmers make a copy of the PI and change the PI to PR to create their prototypes. These prototypes are stored in common libraries that other programmers can access to copy the prototypes used in calling programs. In many cases, a programmer simply uses /COPY to copy the prototype into the calling program.

Some shops even store prototypes as a group and the whole group is copied into the calling program. The compiler does not care about unreferenced prototypes in a calling program.

# Read-only parameters

IBM i

- Specified by CONST keyword
- Implies that called program does not modify the parameter field
  - Parameter treated as read-only
- Compiler to generate temporary fields as necessary:
  - Such as when data type and size do not exactly match
  - Can pass result of expression as a parameter

```

000100  D-----DName+++++ETDsFrom+++To/L+++IDc .Keywords+++++
000200  D TaxCalc          PR                                ExtPgm( 'PAYPROG5' )
000300  D Grossay          8S 2 CONST
000400  D Allowances        8S 2 CONST
000500  D Gross            S       7P 2
000600  D Base             S       6P 2
000700  D PersAllow        S       5P 4
000800
000900
001000 /Free
001100 CallP TaxCalc(Gross : Base + PersAllow);
001200 :
001300 /End-free

```

Means

OW	Column	Replace
0001	1	Temp1 = Gross;
0002	2	Temp2 = Base + PersAllow;
0003	3	CallP TaxCalc(Temp1 : Temp2);

© Copyright IBM Corporation 2009

Figure 8-9. Read-only parameters

AS075.0

## Notes:

The **CONST** keyword tells the compiler to accommodate possible mismatches in the definition of the parameters in the called program versus the calling program.

For example, assume that:

- The program you are calling expects a packed decimal value of five digits with no decimal places.
- The field you actually pass as a parameter is a 3-digit signed numeric.

When you use the **CONST** keyword, you are telling the compiler that as necessary, you want it to make a copy of the data prior to passing it to accommodate any such mismatches. Thus, you avoid the need to explicitly define a working variable of the correct size and type.

The compiler can accommodate differences in:

- **Numeric fields:** size, decimal places, and type (zoned, packed, integer, and so on)
- **Date fields:** format of the date data type field

- **Character fields:** length and type (fixed or varying)

The **CONST** keyword also allows the result of an expression to be passed as a parameter.

## Read-only parameters: Caveat

IBM i

- Called program behaves as expected *only* if CONST also in PI.
- Temporary fields generated when data type and length do not match PR.

```

000100      D TaxCalc          PR           ExtPgm('PAYPROG5')
000200      D                           8S 2 CONST
000300      D                           8S 2 CONST
000400
000500      D Gross            S           7P 2
000600      D Allowances        S           8P 2
000700      D GrossPay         S           8S 2
000800
000900      /Free
001000      CallP TaxCalc(Gross : Allowances); // Generates one temporary field
001100      CallP TaxCalc(GrossPay : Allowances); // No temporary fields generated
001200      /End-free

```

© Copyright IBM Corporation 2009

Figure 8-10. Read-only parameters: Caveat

AS075.0

### Notes:

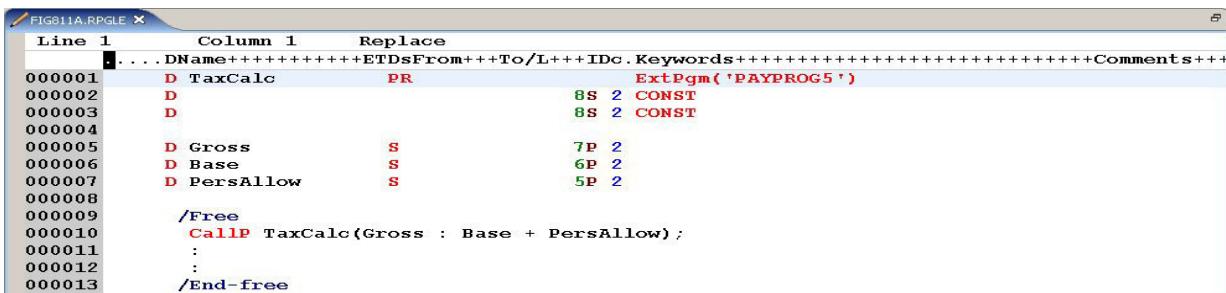
Coding CONST on the prototype of the calling program does not guarantee the field is read-only. Unless you guarantee the read-only status of the parameter by coding the CONST keyword in the Procedure Interface of the called program also, the called program can still change the value of the parameter field.

The RPG IV compiler automatically generates the code needed to perform any required assignment of a parameter field to a compiler generated temporary field for that parameter.

# PR and PI with matching CONST keyword

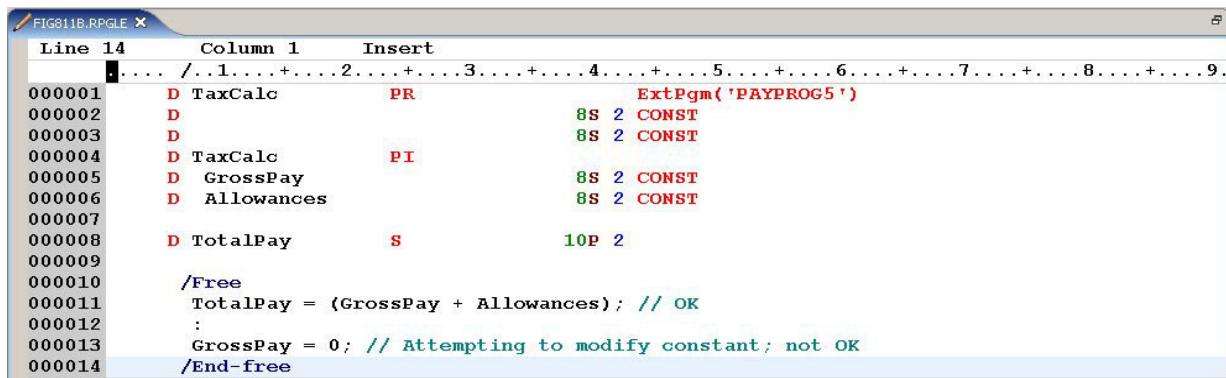
IBM i

## Calling Program



```
FIG811A.RPGLE
Line 1      Column 1      Replace
. .... DName++++++ETDsFrom+++To/L+++ IDC . Keywords+++++Comments+++
000001  D TaxCalc      PR          ExtPgm('PAYPROG5')
000002  D                   8S 2 CONST
000003  D                   8S 2 CONST
000004
000005  D Gross         S          7P 2
000006  D Base          S          6P 2
000007  D PersAllow     S          5P 2
000008
000009  /Free
000010  CallP TaxCalc(Gross : Base + PersAllow);
000011  :
000012  :
000013  /End-free
```

## Called Program



```
FIG811B.RPGLE
Line 14     Column 1      Insert
. .... /...1....+....2....+....3....+....4....+....5....+....6....+....7....+....8....+....9.
000001  D TaxCalc      PR          ExtPgm('PAYPROG5')
000002  D                   8S 2 CONST
000003  D                   8S 2 CONST
000004  D TaxCalc      PI
000005  D GrossPay     S          8S 2 CONST
000006  D Allowances    S          8S 2 CONST
000007
000008  D TotalPay     S          10P 2
000009
000010  /Free
000011  TotalPay = (GrossPay + Allowances); // OK
000012  :
000013  GrossPay = 0; // Attempting to modify constant; not OK
000014  /End-free
```

© Copyright IBM Corporation 2009

Figure 8-11. PR and PI with matching CONST keyword

AS075.0

## Notes:

If you use a the CONST keyword in the Procedure Interface of the called program as well as in the Prototype of the calling program, the read-only promise is guaranteed by the compiler.

The compiler issues an error if:

- You attempt to modify the content of the field directly as shown in the example.
- You attempt to obtain its address (by using **%Addr BIF**).

The reason for this is that when you have the address of a variable, you can base another field on it and change the contents of the parameter indirectly.

- You attempt to pass the field as a parameter to another program or procedure where it is not defined as CONST.

While not as direct as using **%Addr**, this is effectively the same situation.

## PR and PI in called program

You must code the PR before the PI in the called program or the compiler will issue the following message:

\*RNF3782 30 - Prototype for main procedure must be defined prior to procedure-interface definition.

# Options keyword for parameter definition

IBM i

- OPTIONS(\*NOPASS):
  - The parameter is optional.
  - If any parameter is specified as \*NOPASS all subsequent parameters in that prototype must also be \*NOPASS.
- OPTIONS(\* OMIT):
  - The parameter can be omitted.
  - The special value \* OMIT is used on the CALLP.
- OPTIONS(\*VARSIZE):
  - Applies to character fields and arrays only.
  - The parameter can be smaller than specified in the prototype.
  - It must be passed by reference (that is, no VALUE keyword).
- OPTIONS(\*TRIM):
  - It can be used with CONST or VALUE parameter.
  - Leading and trailing blanks are removed.
  - If not varying length, then it is padded with blanks on right (or left if OPTIONS(\*RIGHTADJ)).
- OPTIONS(\*RIGHTADJ):
  - It can be used with CONST or VALUE parameter.
  - Parameter value is right adjusted.

© Copyright IBM Corporation 2009

Figure 8-12. Options keyword for parameter definition

AS075.0

## Notes:

The OPTIONS keyword can be used in a prototype with a parameter. Its keywords provide additional facilities:

- **\*NOPASS:** Means that the parameter does not have to be passed on the call. Any parameters following this specification must also be described as \*NOPASS.  
When the parameter is not passed to a program or procedure, the called program or procedure operates as if the parameter list did not include that parameter.  
The called program can determine the number of parameters it received either by using the %PARMS built-in function.
- **\* OMIT:** Is used for parameters that are not mandatory but that occur in the middle of a parameter sequence and therefore cannot be designated as \*NOPASS.  
When the \* OMIT option is specified, the special value \* OMIT is specified instead of that parameter on the call.  
The \* OMIT parameter is counted in the number of parameters passed and the called program will need to test to see if the parameter was actually passed. Any attempt to

reference the parameter when \*OMIT was passed results in an error. There are two ways to test to see whether the parameter was passed:

- Compare the %Addr of the parameter to \*Null. This cannot be used if the parameter was designated as CONST.
- Use the API CEETSTA.

The *RPG Programmers Guide* provides a brief example in Chapter 10, “Calling Programs and Procedures.”

\*OMIT is only allowed for parameters that are passed by reference, including those that specify the CONST keyword.

- **\*VARSIZE:** Tells the compiler to accept a character parameter that is shorter in length than the prototype specifies. Effectively, it indicates that the length specified is to be treated as a maximum.

This option is often used when the length of the passed field is also passed to the called program (for example, QCMDDEXC).

- **\*RIGHTADJ:** When specified for a CONST or VALUE parameter in a function prototype, the parameter value is right adjusted. This keyword is not allowed for a varying length parameter within a procedure prototype.
- **\*TRIM:** When specified for a CONST or VALUE parameter of type character, the passed parameter is copied without leading and trailing blanks to a temporary work variable. If the parameter is not a varying length parameter, the trimmed value is padded with blanks (on the left if OPTIONS(\*RIGHTADJ) is specified; otherwise, on the right). Then the temporary is passed instead of the original parameter.

# Example prototype for QCMDEXC

IBM i

- This prototype allows you to:
  - Use a more meaningful name - OvrDBFile
  - Use a variable size parameter - Options(\*VarSize)
  - Avoid having to move values to 15,5 fields - Const and omit parameters - Options(\*NoPass)

```

FIG813.RPGLE x
Line 18   Column 1   Insert
. .... /....+....2....+....3....+....4....+....5....+....6....+....7....+....8....+....9.
000001  D CusMastOvr    S          40A
000002  D FileName       S          10A  Inz('ITEM_PF')
000003  D
000004  D OvrDBFile      PR          ExtPgm('QCMDEXC')
000005  D CmdString       3000  Options(*VarSize) Const
000006  D CmdLength        15P 5 Const
000007  D CmdOpt          3     Options(*NoPass) Const
000008
000009  /FREE
000010
000011  CusMastOvr = 'OVRDBF FILE(' + %TrimR(FileName) + ') EOFDLY(600)';
000012
000013  OvrDBFile(CusMastOvr: %Len(CusMastOvr));
000014
000015  OvrDBFile('OVRDBF FILE(' + %TrimR(FileName) + ') EOFDLY(600)': 40);
000016
000017  *InLr = *On;
000018  /END-FREE

```

© Copyright IBM Corporation 2009

Figure 8-13. Example prototype for QCMDEXC

AS075.0

## Notes:

This example of a prototype for QCMDEXC illustrates the following features of prototyping:

- You use a prototype name indicative of the task, **OvrDBFile**, as an alias for QCMDEXC.
- You can specify **Options(\*VarSize)**, and therefore pass only the 40-character parameter *CusMastOvr* rather than the 3,000 character length specified in the prototype.
- You do not have to create a work field (of 15,5) to pass an integer.
- You can omit the third parameter of QCMDEXC.

QCMDEXC has a third parameter that is used by DBCS (Double Byte Character Set) support.

# What else can prototypes do?

IBM i

- They are the key to many of the APIs shipped with the iSeries.
  - The C function library, which includes:
    - Random number generation
    - Math functions (Sin, Cos, Tan, and so forth)
    - MI instructions (MI programming in RPG!)
    - Much more... .
  - TCP/IP functions
    - You can use TCP/IP sockets functions directly from RPG.
  - ILE APIs
    - For dynamic memory, error handling and more
  - System APIs
    - Most system APIs are written with the C programmer in mind.
      - For example, the Domino Hi-Test APIs
- The RPG Redbook contains examples.
  - *Who Knew You Could Do That with RPG IV? - SG24-5402*



© Copyright IBM Corporation 2009

Figure 8-14. What else can prototypes do?

AS075.0

## Notes:

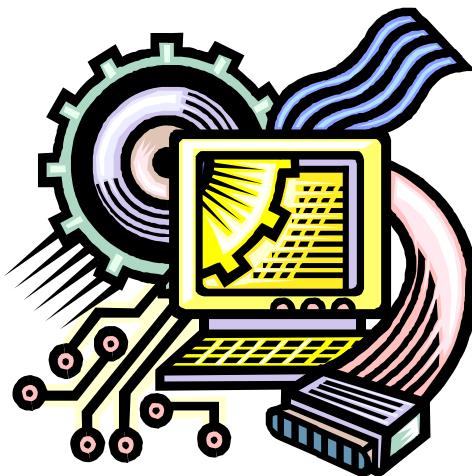
Prototypes can also be used to call procedures that are written in other languages, most notably C. This means that RPG IV programs have access to all the functions in the C function library, which is shipped as part of i5/OS (OS/400) on all systems. Other system APIs (Application Program Interfaces) previously available only to C programmers, such as TCP/IP sockets and direct program access to the IFS, are also enabled by the prototyping support associated with subprocedures.

As an alternative to QCMDEXC, through the power of prototyping, the C *system* function is available. This facility has advantages over using QCMDEXC because it allows you to determine the message number when the command fails. The QCAPCMD API is preferred over both QCMDEXC and *system()* because of its better feedback and error handling.

Read *Who Knew You Could Do That With RPG IV?*, Chapter 5, “Exploring new ways to exploit your AS/400 system.”

# Machine exercise: Prototyping

IBM i



© Copyright IBM Corporation 2009

Figure 8-15. Machine exercise: Prototyping

AS075.0

## Notes:

Perform the machine exercise “Prototyping.”

## Unit summary

IBM i

Having completed this unit, you should be able to:

- Create a prototype for calling a program
- Create a procedure interface for a program
- Explain the advantages of using prototypes

© Copyright IBM Corporation 2009

---

Figure 8-16. Unit summary

AS075.0

### Notes:

# Unit 9. Using subprocedures

## What this unit is about

This unit describes how to code and call RPG IV subprocedures.

## What you should be able to do

After completing this unit, you should be able to:

- Code an RPG IV subprocedure
- Code a prototype for a subprocedure
- Code a procedure interface for a subprocedure
- Code a subprocedure that returns parameters
- Code and call a subprocedure that returns a value

## How you will check your progress

- Machine exercise

# Unit objectives



IBM i

After completing this unit, you should be able to:

- Code an RPG IV subprocedure
- Code a prototype for a subprocedure
- Code a procedure interface for a subprocedure
- Code a subprocedures that returns parameters
- Code and call a subprocedure that returns a value

© Copyright IBM Corporation 2009

---

Figure 9-1. Unit objectives

AS075.0

## Notes:

# What is a subprocedure?

IBM i

- *Subprocedure* is the RPG IV name for user-defined procedure.
  - Evolution of subroutines
- Subprocedures can:
  - Define their own local variables
    - Only the code associated with the variable can change its content.
  - Access global variables defined in the main body of the source
  - Access any files defined in the program
  - Be called recursively
- The compilation unit can have any number of subprocedures.
  - Each subprocedure must have its own prototype.
- Compilation unit is source code processed by the compiler in a single compilation.
  - Includes any /COPY members

© Copyright IBM Corporation 2009

Figure 9-2. What is a subprocedure?

AS075.0

## Notes:

A subprocedure could be perceived as the modern version of the subroutine. However, whereas subroutines are always *local* to a specific program, subprocedures can be used and called by many programs.

Before we discuss subprocedures further, we should define the term procedure. A *procedure* is the RPG IV name for what we commonly call a program. We refine this definition when we cover ILE but, at this point, think of a procedure as a type of program.

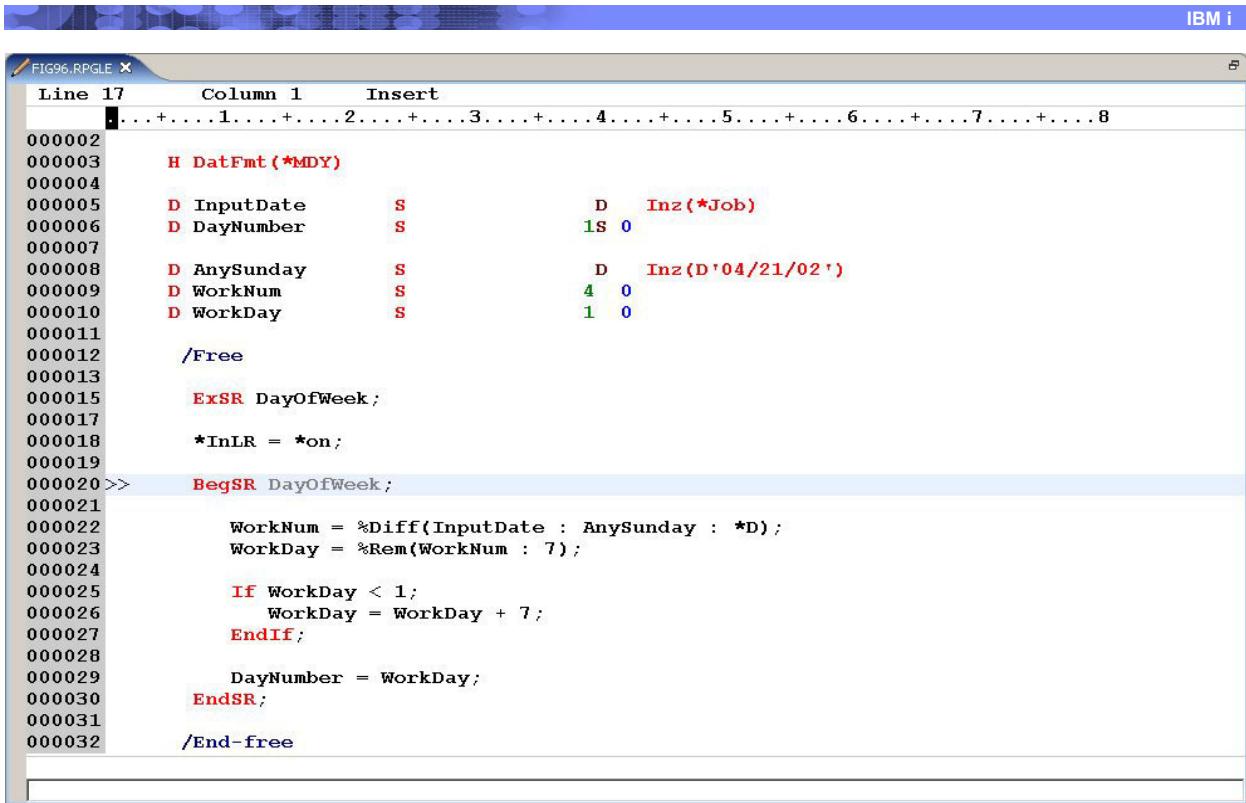
A *subprocedure* is a subprogram that is contained within a procedure or packaged in an ILE Service Program. An RPG IV *subprocedure* is the logical evolution of an RPG subroutine.

User-written subprocedures in RPG IV allow *recursion*. Recursion means that a subprocedure can repeatedly call itself, directly or indirectly, within a job stream. Subprocedures support local variables; that is, fields that are defined within a subprocedure, and are only available to and affected by logic within the bounds of the subprocedure.

RPG IV subprocedures require the use of prototypes. In this topic, we use prototypes in RPG IV subprocedures. However, you find later that many of the same prototype-writing skills can be applied to access system APIs and C functions.

By *compilation unit*, we include the RPG IV source that can be successfully processed by the RPG compiler. In RPG IV, each *compilation unit* is compiled into a module (**\*MODULE**) object. Several modules can be grouped into a single program.

## Example: DayOfWeek subroutine



The screenshot shows an IBM i terminal window with the title bar "FIG96.RPGLE X" and the status bar "IBM i". The main area displays RPGLE code for a subroutine named "DayOfWeek". The code includes declarations for InputDate, DayNumber, AnySunday, WorkNum, and WorkDay, and logic to calculate the day number based on the input date and any Sunday value. The code is annotated with comments and assembly language-like syntax.

```

Line 17      Column 1      Insert
000002
000003      H DatFmt (*MDY)
000004
000005      D InputDate      S          D   Inz(*Job)
000006      D DayNumber      S          1S  0
000007
000008      D AnySunday      S          D   Inz(D'04/21/02')
000009      D WorkNum        S          4  0
000010      D WorkDay        S          1  0
000011
000012      /Free
000013
000015      ExSR DayOfWeek;
000017
000018      *InLR = *on;
000019
000020 >>    BegSR DayOfWeek;
000021
000022      WorkNum = %Diff(InputDate : AnySunday : *D);
000023      WorkDay = %Rem(WorkNum : 7);
000024
000025      If WorkDay < 1;
000026          WorkDay = WorkDay + 7;
000027      EndIf;
000028
000029      DayNumber = WorkDay;
000030      EndSR;
000031
000032      /End-free

```

© Copyright IBM Corporation 2009

Figure 9-3. Example: DayOfWeek subroutine

AS075.0

### Notes:

In this visual, we are showing you an existing subroutine. In the following examples, we create a subprocedure from this particular subroutine. Existing subroutines in RPG IV programs are excellent candidates for subprocedures. You learn how similar subprocedures and subroutines are as we progress through this example.

The code example shows the traditional use of a standard subroutine, including its inherent problems. **WorkNum** and **WorkDay** are *standard* field names within the subroutine that we are referencing. They are defined in our main procedure yet are used as *local* work variables within the subroutine.

**InputDate** and **DayNumber** are being used as if they are parameters — a good practice for isolating the mainline logic from the subroutine logic.

The use of subroutines forces us to use naming standards to ensure that work fields within the subroutine do not get used in the mainline logic. However, *all* variables are defined globally for the program. Only our coding conventions and standards enables us to isolate the scope and use of certain variables. These conventions cannot be enforced, and might easily be overlooked or ignored during later maintenance.

# Example: Basic subprocedure

```

FIG92B.RPGLE X
Line 15      Column 1      Insert
. . . PName+++++. . . B.....Keywords+++++. . . Comments+++++
000001      //***** Main Procedure
000003      H  DatFmt(*MDY)  DftActGrp(*No)
000005      D  DayOfWeek    PR
000007      D
000008          D
000009          1S 0
000010      D InputDate     S      D  Inz(*Job)
000012      D DayNumber     S      1S 0
000014      /Free
000015          CALLP DayOfWeek(InputDate:DayNumber);
000016          *InLR = *on;
000017      /End-free
000019
000020
000021      //***** Subprocedure (inline)
000022      P DayOfWeek     B
000023
000024      D DayOfWeek     PI
000025          D
000026          1S 0
000027
000028      D AnySunday    S      D  Inz(D'04/21/02')
000029      D WorkNum      S      4  0
000030      D WorkDay      S      1  0
000031      /Free
000032          WorkNum = %Diff(WorkNum : AnySunday : *D);
000033          WorkDay = %Rem(WorkNum : 7);
000034          IF WorkDay < 1;
000035              WorkDay = WorkDay + 7;
000036          EndIf;
000037          DayNo = WorkDay;
000038          Return;
000039      /End-Free
000040      P DayOfWeek     E
000041

```

© Copyright IBM Corporation 2009

Figure 9-4. Example: Basic subprocedure

AS075.0

## Notes:

This is the equivalent subprocedure based on the subroutine.

Notice the sequence of the specifications that are in the visual. In this example, the subprocedure is coded *in line* following the main procedure. We cover the main components of the process step by step in subsequent visuals.

In summary:

- The prototypes for the subprocedure are coded first. The main procedure calls the subprocedure. Therefore, a PR that matches the PI for the subprocedure is required.
- The EXSR has been replaced with a Call to the subprocedure using the familiar CALLP operation.
- BEGSR is replaced by a new P-specification that marks the beginning of subprocedure with the character 'B' in the appropriate column. P-Specs appear *after* the regular C specs. We look in more detail at the sequence of specifications in this "new style" of RPG program later.
- The D-specs for fields used only by the subprocedure are coded next.

The three fields with an **S** (stand-alone fields) are local variables available only to the logic in this particular subprocedure.

- The logic functions performed by the subprocedure are coded next.
- A Return is coded in order to pass control back to the caller of the subprocedure.
- A P-spec is coded in order to end the subprocedure. This designates the end of the subprocedure.

# Different types of RPG IV source members

- Traditional style
  - Main line only
- RPG IV main + subprocedures:
  - Main procedure:
    - Define files and global variables
    - C specs define mainline logic
    - Uses RPG logic cycle
  - Local subprocedures:
    - Can define their own (local) variables
    - Can access (global) variables in main procedure
    - No RPG logic cycle
- RPG IV subprocedures only (recommended)
  - Main section can define files and variables but no logic:
    - No main procedure; NOMAIN keyword
    - No RPG logic cycle

© Copyright IBM Corporation 2009

Figure 9-5. Different types of RPG IV source members

AS075.0

## Notes:

When you want to use the features of subprocedures, you have two choices:

1. You can include the subprocedure code in the same source member as the mainline code that calls it.

When you do include subprocedures in the same source member as the mainline, you have coded a single main procedure, followed by one or more subprocedures. The main procedure defines all files and global data. Global data items are usable by any subprocedures coded in the same module. Fields in files are always considered global, because all files must be defined in the main procedure and data defined in the procedure are global within the module.

2. You can place your subprocedures in a separate module, using the *cycle-less* support mentioned in the visual. Subprocedures can be grouped as single or multiple subprocedures in a NOMAIN procedure.

This is the recommended method of coding subprocedures. They are packaged in a much more *modular* fashion. They are easier to maintain and to modify. They can be easily called by those programs that you authorize to access them.

RPG IV subprocedures must be coded specifying no logic cycle (*cycle-less*). This is because the RPG cycle code is not generated by the compiler, because it is for all RPG IV modules with a mainline. The benefit of *cycle-less* coding is that calls to subprocedures in these *cycle-less* modules are very fast because there is no logic cycle overhead.

When you first try to create subprocedures, you might receive the compiler message:

RNF3788 - Keyword EXTPGM must be specified when DFTACTGRP(\*YES) is specified on the CRTBNDRPG command.

You might receive this message because the default values on the CRTBNDRPG command assume that the type of compilation you want is for an OPM program.

Subprocedures are an integral part of the ILE environment. If you are using subprocedures, you are using a feature of ILE. The default values must be changed. With the CODE editor, you can change the defaults for the command on your PC and leave the default on the i5(iSeries) as-is.

### Compiling with Subprocedures

You must compile with the option **DFTACTGRP(\*NO)**.

# Major features of subprocedures

- Compilation unit can contain many subprocedures.
- Subprocedures can:
  - Be called by CALLP or from expression
  - Call other subprocedures
  - EXSR to inline subroutines in subprocedure
  - Return parameters (CALLP)
  - Return a value (to expression), such as RPG IV BIFs
- Subprocedures cannot:
  - Contain coding of other nested subprocedures
  - EXSR to subroutines in MAIN procedure
- Subprocedures can return a value:
  - Called like a function, such as RPGIV BIFs

```
....+...1....+...2....+...3....+...4....+...5....+...6....+...7....+
0001 /Free
0002 >>1 IF DayOfWeek(ADateField) > 5;
0003 |
0004 >>2 WeekDay = DayOfWeek(Today);
0005 EndIF;
0006 /End-Free
```

© Copyright IBM Corporation 2009

Figure 9-6. Major features of subprocedures

AS075.0

## Notes:

Subprocedures that return a value can be used in a manner similar to RPG IV built-in functions, as shown in this example. In this example, we are calling a subprocedure **DayOfWeek**, which requires a single input parameter.

1. In the first line of code in the visual, the parameter is the field named **ADateFld**.
2. In the second line of code, the parameter is the field **Today**.

**DayOfWeek** returns a value, the day of the week (1 through 7 representing Sunday through Saturday).

The returned value replaces the function call in the statement. In the second line of code, the returned value is placed in the field **WeekDay**.

Notice the use of a subprocedure in a free form expression.

The free form call statement, **CALLP** (Call with Prototype), can be used to invoke any subprocedure that does not return a value.

# What are local variables?

```

FIG95.RPGLE x
Line 12      Column 1      Insert
.....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8....+....9....+....0
000001 // Main Procedure
000002 D Count      S          5P 0 Inz
000003 D Temp       S          20A
000005 /FREE
000006 Count = Count + 1;           // OK
000007 LocalValue = 0;             // Wrong - local to Subprocedure (1)
000008 Temp = 'Temp in Main';     // OK
000009 *InLR = *On;
000010 /END-FREE
000011 // End Main Procedure
000012
000013 // Begin Subprocedure (1)
000014 PSubproc1    B
000015 D LocalValue S          7P 2 Inz
000016 D Temp       S          7P 2 Inz
000018 /FREE
000019 Count = Count + 1;           // OK
000020 LocalValue = 0;             // OK
000021 Temp = LocalValue;         // OK
000022 *InLR = *On;
000023 /END-FREE
000024 P             E
000025 // End Subprocedure (1)
000027 // Begin Subprocedure(2)
000028 PSubproc2    B
000029 D Temp       S          40A
000031 /FREE
000032 LocalValue = 0;             // Wrong - local to Subprocedure (1)
000033 Temp = 'Temp in Subprocedure2'; // OK
000034 *InLR = *On;
000035 /END-FREE
000036 P             E
000037 // End Subprocedure (2)

```

© Copyright IBM Corporation 2009

Figure 9-7. What are local variables?

AS075.0

## Notes:

Any subprocedures that are coded inline within the source member that contains the main procedure automatically can access global data items defined in that main procedure.

These subprocedures can also include D-spec definitions of their own local data fields. These local data fields are accessible only within the subprocedure in which they are defined.

**Import** and **Export** keywords can be used within subprocedures to share data with other subprocedures or the mainline, if desired. Also, parameters can be passed back and forth as well, as defined in the prototype. We will discuss **Import** and **Export** for data in the next unit.

In the example, the P-specs and PI and PR are not shown. Focus your attention on local versus global variables.

# Basic subprocedure

IBM i

FIG97.RPGLE X

Line	Column 1	Replace
000001	// Main Procedure	
000003	H DatFmt(*MDY)	
000005 > 5	/Copy DOWProto	
000007		
000008	D InputDate S	D Inz(*Job)
000012	D DayNumber S	1 0
000013		
000014	/Free	
000015 > 1	DayNumber = DayofWeek(InputDate);	
000017		
000018	*InLR = *on;	
000019	/End-free	
000020		
000021	// Subprocedure (inline)	
000022 > 2	DayOfWeek B	
000023	D DayOfWeek PI	1S 0
000024	D InpDate	D
000025		
000026	D AnySunday S	D Inz(D'04/21/02')
000027	D WorkNum S	4 0
000028 > 3	WorkDay S	1 0
000029		
000030	/Free	
000031	WorkNum = %Diff(InpDate : AnySunday : *D);	
000032	WorkDay = %Rem(WorkNum : 7);	
000033		
000034	If WorkDay < 1;	
000035	WorkDay = WorkDay + 7;	
000036	EndIf;	
000037		
000038 > 4	Return WorkDay;	
000039	/End-Free	
000040	P DayOfWeek E	

**Figure 9-8.** Basic subprocedure

AS075.0

## **Notes:**

This is a further refinement of the subprocedure we saw earlier, based on the subroutine. We have used the strongly recommended practice of coding /COPY to bring in the prototype code.

The **/COPY DOWPROTO** line of code copies the source member that contains the prototype code. This is very similar to the program prototypes that we discussed earlier. We look at it again in a later visual.

Once again, the subprocedure is coded *in line* following the main procedure. Here is a reminder of the sequence of specifications:

5. The prototypes for the subprocedure are coded first. The main procedure calls the subprocedure. Therefore, a PR that matches the PI for the subprocedure is required.

Any D-specs for the main procedure follow the PR.

1. Call of subprocedure from an expression (like a BIF).

2. The P-specification that begins the subprocedure appears after the regular C specs. We look in more detail at the sequence of specifications in this *new style* of RPG program later.

The D-specs for fields used only by the subprocedure are coded next. We no longer require the **DayNo** variable from our earlier example because we are returning the result on the RETURN operation (explained as follows).

3. The three fields with an **S** (stand-alone fields) are local variables available only to the logic in this particular subprocedure.

Logic for the subprocedure.

4. A RETURN to the Caller. This statement is mandatory in this example because we are returning a value from the subprocedure to the caller.

A P-spec to End the subprocedure.

# Invoking the subroutine

The diagram illustrates the replacement of a subroutine call with a Built-In Function (BIF)-like expression. At the top, a code editor window shows lines 00012 through 00014 of an RPG program:

```

00012     WorkDate = InputDate;
00013     ExSR DayOfWeek;
00014     WorkDay = DayNum;

```

An orange arrow points from this code down to another code editor window at the bottom. The bottom window shows lines 00008 through 00015. Line 00015 contains a BIF-like expression:

```

00008     .1.....+....2.....+....3.....+....4.....+....5.....+....6.....+....7.....+
00009     D InputDate      S
00010     D DayNumber       S
00011           1   8
00012
00013
00014 /Free
00015>>1 DayNumber = DayOfWeek(InputDate);

```

A large orange arrow points from the original code to the BIF-like expression. A callout bubble labeled "Replace with" points to the arrow.

Call to DayOf Week subroutine is BIF-like.

Field InputDate is passed as a parameter to the subroutine.

© Copyright IBM Corporation 2009

Figure 9-9. Invoking the subroutine

AS075.0

## Notes:

1. A subroutine is called in much the same syntax as you call a built-in function in RPG. Because this subroutine returns a value, it is called from an expression. The returned day number value is placed in the field **DayNumber**.

If the subroutine was not coded to return a value, we would invoke it with a CALLP.

# P-specs and the procedure interface

IBM i

- Subprocedure delimited by P-specs
  - B(egin) names the procedure
  - E(nd) required to complete it
- Procedure interface required
  - PI optionally defines data type and length of return value
  - Code that follows define any parameters
    - Parameters terminated by non-blank entry in col 24-25 of D-Spec

The procedure name is optional on both the PI and the E(nd) P Spec

Line No.	Column No.	Value	Notes
00022	2	P DayOfWeek	B
00023		D DayOfWeek	PI
00024		D InpDate	
00025		:	
00040		:	
00041		P DayOfWeek	E

© Copyright IBM Corporation 2009

Figure 9-10. P-specs and the procedure interface

AS075.0

## Notes:

Subprocedures begin with a P-spec and must be terminated by a P-spec.

In this example, you can see both the beginning P-spec and the ending P-spec. The beginning P-spec contains a B in the column position that would contain, for example, a DS on a D-spec. The P-spec is formatted very similar to the format of the D-spec.

The next thing we need is a Procedure Interface, or PI. The procedure interface defines the interface to the procedure -- most significantly, the parameters passed to the subprocedure, from the subprocedure, or both. The PI is defined on the D-specs, typically as the first D-specs in the subprocedure.

The data item defined on the same line as the PI is the return value.



**Note**

It is possible to have a subprocedure that returns *no* value. A subprocedure can return, at most, one value.

The data items that follow the PI with nothing specified in the column below the PI-spec are input parameter to the subprocedure. In this example, the **InpDate** field is the only parameter.

# Local variables in DayOfWeek subprocedure

IBM i

		1.....+.....2.....+.....3.....+.....4.....+.....5.....+.....6.....+.....7.....+.....8.....	
000023	D DayOfWeek	PI	1 S 0
000024	D InpDate		D
000025			
000026	D AnySunday	S	D Inz(D '04/21/02')
000027	D WorkNum	S	4 0
000028>>③	D WorkDay	S	1 0

- Local variables:
  - Define in subprocedure
  - Belong only to subprocedure
  - Cannot be changed outside subprocedure
  - Can reference only in subprocedure where defined
- Stand-alone field **Any Sunday** terminates PI.

© Copyright IBM Corporation 2009

Figure 9-11. Local variables in DayOfWeek subprocedure

AS075.0

## Notes:

The data items following the single parameter, **InpDate**, are fields that are *local to the subprocedure*. The procedure interface is terminated by a D-spec with an entry in the column where PI was specified (in this example, an **S**).

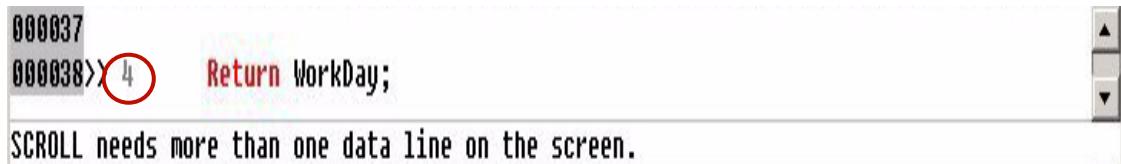
Local data is significant in your implementation of code that is reusable. It is also important to make maintenance tasks less prone to error, because there is more variable independence within the subprocedures and, therefore, less chance that maintenance tasks will inadvertently affect your logic.

In this example, the fields **AnySunday**, **WorkNum**, and **WorkDay** are fields that are *local to this subprocedure*. They can be referenced only within the subprocedure.

## Returning the result

IBM i

- Can return a simple value:



```
000037
000038)X 4 Return WorkDay;
SCROLL needs more than one data line on the screen.
```

- Can return an expression:



```
000035)X 4 Return WorkDay + 7;
```

© Copyright IBM Corporation 2009

Figure 9-12. Returning the result

AS075.0

### Notes:

Here we specify the return value for this subprocedure. We use the **RETURN** operation code and specify the return value.

The returned value can be either a single field or an expression, as in the second example. In fact, because a subprocedure can be used anywhere a variable of its type can be used, the returned value itself could be the result of a subprocedure invocation. But we do not discuss recursion here.

## Defining the prototype

IBM i

- Required in caller for each subprocedure called
- Almost identical to PI
- Used to validate parameters during compile
- Create from PI
- COPY into procedure

```
|00100  D DayOfWeek      PR
|00200  D
```

```
1S 0
D
```

© Copyright IBM Corporation 2009

Figure 9-13. Defining the prototype

AS075.0

### Notes:

The next step is to define the prototype. As discussed earlier, the parameters in the prototype must match the Procedure Interface because it defines the interface to the procedure.

The prototype is used by all procedures that call this subprocedure. The prototype must also be coded in the module where the procedure is defined. This is done so that the compiler can check the validity of the prototype; that is, it checks that the parameters specified match the Procedure Interface in parameter number and type. This means that if the subprocedure is placed in the same source member as the caller (as in this example), then only a single copy of the prototype is required, because the compiler will be able to check the prototype and procedure interface in a single compile step. If the subprocedure were to be placed in a separate source member, then a copy of the prototype would be required in both the member containing the subprocedure and in the main procedure (or calling procedure) as well as in any other main or subprocedures calling this subprocedure.

A common, and encouraged, practice is to place the prototypes for subprocedures in a separate source member that is copied in via the /COPY directive. This is especially

important if the subprocedure is coded as a separate source member and then created as a separate module. The prototype in the calling procedure *must* match the one in the defining procedure. This prototype is the one in the module containing the subprocedure that the compiler verified for you.

# RPG IV specification sequence

IBM i

H	Keyword NOMAIN must be used if there are no main line calculations.
F	File Specifications – always global
D PR	<b>Prototypes for all procedures used OR defined in the source (often present in the form of a /COPY)</b>
D ProgramName PR	<b>Only used if a Procedure Interface (below) is being used to replace the program's *ENTRY PLIST</b>
D ProgramName PI	<b>Used as an alternative to an *ENTRY PLIST</b>
D	<b>Data definitions – global</b>
I	<b>GLOBAL</b>
C	<b>Main calculations (Any subroutines are local)</b>
O	<b>GLOBAL</b>
P ProcName1 B	<b>Start of first procedure</b>
D PI	<b>Procedure Interface</b>
D	<b>Data definitions – local</b>
C	<b>Procedure calcs (Any subroutines are local)</b>
P ProcName1 E	<b>End of first procedure</b>
P Proc..... B	<b>Start of next procedure</b>
.....	<b>Procedure interface, D-specs, C-specs, and so forth</b>
P Proc..... E	<b>End of procedure</b>
**	<b>Compile time data</b>

© Copyright IBM Corporation 2009

Figure 9-14. RPG IV specification sequence

AS075.0

## Notes:

This visual illustrates the layout of a hypothetical *complete* RPG IV program containing one or more subprocedures.

Notice that the **NOMAIN** keyword on the H-specification is optional and indicates that there is no mainline logic in this module, that is, no C-specs outside the subprocedure logic. Notice also that any F-specs always go at the top of the member, just after the H-spec, for any files that are accessed, either by the mainline code (if any) or by the subprocedures. This is true regardless of whether or not there is any mainline logic.

The first PI line in the program is serving as a replacement for the \*ENTRY LIST for this program or main procedure module. This is optional.

The D- and I-specs that follow are for data items in the mainline, which are global; that is, they can be accessed from both mainline logic and any subprocedures in this module.

Following the O-specs for the mainline code is the beginning P-spec for the first subprocedure. It is followed by the PI (procedure interface) for that subprocedure. D and C specs for this subprocedure are next, followed by the ending P-spec.

Any other subprocedures would follow these, each with its own set of beginning and ending P-specs.

# Subprocedures calling other subprocedures

IBM i

- For a specific date, determine day ("Monday", "Tuesday", and so forth)

```

-----+-----+-----+-----+-----+-----+-----+-----+
000001  P DayName      B
000002
000003  D              PI      9
000004  D  InpDate      D
000005
000006  D DayData       DS
000007  D              DS
000008  D              63     Inz('Monday   Tuesday   Wednesday+
000009          Thursday Friday   Saturday +
000010          Sunday    ')
000011  D              9     Overlay(DayData) Dim(7)
000012
000013  D WorkDay       S      1  0  Inz
000014
000015  /Free
000016
000017  WorkDay = DayOfWeek(InpDate);
000018  Return DayArray(WorkDay);
000019
000020  /End-Free
000021  P DayName      E
-----+-----+-----+-----+-----+-----+-----+-----+

```

© Copyright IBM Corporation 2009

Figure 9-15. Subprocedures calling other subprocedures

AS075.0

## Notes:

This visual illustrates how subprocedures can call other subprocedures. The new DayName subroutine calls the DayofWeek procedure to get the number of the day of the week. The DayName procedure then translates the number into a day name.

Note that the /COPY member for the prototypes in this case would need to include a prototype for DayName and one for DayofWeek, because DayName calls DayofWeek.

# An alternate approach

IBM i

This:

```

000013 D WorkDay      S          1  0 Inz
000014
000015   /Free
000016
000017   WorkDay = DayOfWeek(InpDate);
000018   Return DayArray(WorkDay);
000019
000020   /End-Free

```

Could be:

```

....+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
000015   /Free
000016
000018   Return DayArray(DayOfWeek(InpDate));
000019

```

No WorkDay variable required!

© Copyright IBM Corporation 2009

Figure 9-16. An alternate approach

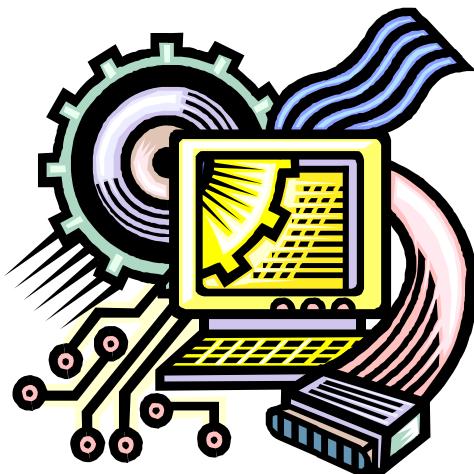
AS075.0

## Notes:

Because subprocedures that return a value can be used anywhere a field name or constant of that type (that is, alphanumeric or numeric) can be used, the second example you see here could be used to incorporate a more efficient programming style.

# Machine exercise: Subprocedures

IBM i



© Copyright IBM Corporation 2009

Figure 9-17. Machine exercise: Subprocedures

AS075.0

## Notes:

Perform the machine exercise “Subprocedures.”

## Unit summary

IBM i

Having completed this unit, you should be able to:

- Code an RPG IV subprocedure
- Code a prototype for a subprocedure
- Code a procedure interface for a subprocedure
- Code a subprocedure that returns parameters
- Code and call a subprocedure that returns a value

© Copyright IBM Corporation 2009

---

Figure 9-18. Unit summary

AS075.0

### Notes:

# Unit 10. An introduction to the Integrated Language Environment

## What this unit is about

This unit describes how to create ILE objects and how to use them in your applications. This unit defines ILE terms as well as describes the relationship of ILE objects and how they are packaged.

ILE enables you to implement modular coding techniques. This unit provides you with the basic concepts and you perform several simple exercises to assist you in understanding how applications are packaged in the Integrated Language Environment.

## What you should be able to do

After completing this unit, you should be able to:

- Describe the benefits of ILE
- Create an ILE module
- Create an ILE service program
- Create an ILE program using static binding

## How you will check your progress

- Machine exercise

## References

SC41-5606      *i5(iSeries) ILE Concepts*

# Unit objectives

IBM i

After completing this unit, you should be able to:

- Describe the benefits of ILE
- Create an ILE module
- Create an ILE service program
- Create an ILE program using static binding

© Copyright IBM Corporation 2009

---

Figure 10-1. Unit objectives

AS075.0

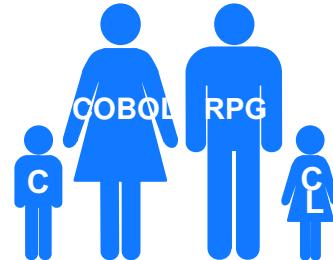
## **Notes:**

## 10.1. Terms and packaging

# What is ILE?

IBM i

- ILE family of compilers:
  - RPG IV
  - ILE C
  - ILE COBOL
  - ILE CL
  - Bundled in 5722-WDS
- System support:
  - Static binding, any-to-any
  - Improves modular designs
  - Better application control via activation groups
  - Coexistence with existing (non-ILE) applications
- A superior way to develop applications



© Copyright IBM Corporation 2009

Figure 10-2. What is ILE?

AS075.0

## Notes:

ILE, the *Integrated Language Environment*, is built into i5/OS (OS/400). However, it can only be used with the compilers that can generate ILE objects.

*Static binding* can be done between any and all of the ILE languages. We will contrast static binding with traditional dynamic binding that is performed when you execute a dynamic call (**CALL** opcode or **CALLP** with PR that specifies **EXTPGM**).

Better application control comes when you use **Activation Groups**, a run-time ability to subdivide the resources for a job into distinct groups, typically applications. We cover Activation Groups in the Advanced Workshop of this series.

Programs that are *not* ILE programs are called OPM programs, or Original Program Model programs.

# Why ILE?

IBM i

- Modularize applications to component level:
  - Easier to maintain and enhance
  - Adapt for changing business requirements
  - Reuse components in new applications
- Applications can incorporate new technologies:
  - Future applications; separate user interface, logic, DB access
  - Creation of services that can be called from any language
- Create your own BIFs.
- Package reusable functions into service programs containers.
- Use best language for the job.
- Use ILE error handling.
- Activation groups give better control of jobs.

**Position your applications for the future!**

© Copyright IBM Corporation 2009

Figure 10-3. Why ILE?

AS075.0

## Notes:

ILE is a tool to use as you create applications that are positioned for the future.

You want to create applications that can be easily enhanced and maintained.

You want applications that are flexible in their ability to incorporate new technologies. Your future applications need to have separated user interface, logic and database access. You want to create services that can be called from any language. If your logic is separate from your user interface, you can have a 5250 front end as well as a Java client on the PC. If your database access is separated from your program, it can be called from multiple front ends. ILE gives you the tools you need to create these types of applications in a more effective manner than traditional methods.

Using ILE module support and the RPG IV subprocedure support, you can start creating your own built in functions. You can package reusable function into service programs.

With ILE you can easily mix and match languages, using the best language for the job. If you do mix languages, you can use ILE error handling APIs for consistent error handling across the languages.

If you need better commitment control, activation groups ensure that data is not committed until your application says to commit.

One of the major benefits of ILE is modularity.

Following are benefits from using a modular approach to application programming:

- Faster compile time

The smaller the piece of code you compile, the faster the compiler can process it. This benefit is particularly important during maintenance, because often only a line or two needs to be changed. When you change two lines, you might have to recompile 2000 lines.

If you modularized the code to take advantage of the binding capabilities of ILE, you may need to recompile only 100 or 200 lines. Even with the binding step included, this process is considerably faster.

- Simplified maintenance

When updating a very large program, it is very difficult to understand exactly what is going on. This is particularly true if the original programmer wrote in a style different than your own. A smaller piece of code tends to represent a single function, and it is far easier to grasp its inner workings. Therefore, the logical flow becomes more obvious, and when you make changes, you are far less likely to introduce unwanted side effects.

- Simplified testing

Smaller compilation units encourage you to test functions in isolation. This isolation helps to ensure that test coverage is complete; that is, that all possible inputs and logic paths are tested.

- Better use of programming resources

Modularity lends itself to greater division of labor. When you write large programs, it is difficult (if not impossible) to subdivide the work. Coding all parts of a program may stretch the talents of a junior programmer or waste the skills of a senior programmer.

- Easier migrating of code from other platforms

Programs written on other platforms, such as UNIX, are often modular. Those modules can be migrated to IBM i and incorporated into an ILE program.

Another advantage of ILE is reusable components. ILE allows you to select packages of routines that can be blended into your own programs. Routines written in any ILE language can be used by all i ILE compiler users. The fact that programmers can write in the language of their choice ensures you the widest possible selection of routines.

The same mechanisms that IBM and other vendors use to deliver these packages to you are available for you to use in your own applications. Your installation can develop its own set of standard routines, and do so in any language it chooses.

Not only can you use off-the-shelf routines in your own applications, you can also develop routines in the ILE language of your choice and market them to users of any ILE language.





## 10.2. Creating ILE objects

## Creating non-ILE RPG IV programs (1 of 4)

IBM i

- Edit: Code the program and create a member in QRPGLESRC
- Compile: Use CRTBNDRPG to create a \*PGM object
- Repeat Steps 1 and 2 for each called program
- Execute: CALL program via command line, menu, and so forth
- Created a non-ILE \*PGM = Original Program Model

© Copyright IBM Corporation 2009

Figure 10-4. Creating non-ILE RPG IV programs (1 of 4)

AS075.0

### Notes:

This visual reviews the steps required to create and execute a program. We have been using this process to create programs in the exercises and then to execute them.

When you want to execute a program, you call it from the command line, from a menu or even from another program.

The traditional calls we have been coding are *dynamic* because we bind them dynamically at call time. At call time, the system first attempts to find the called program on the system. Assuming the program is found, the system then initiates it by opening files, and so on.

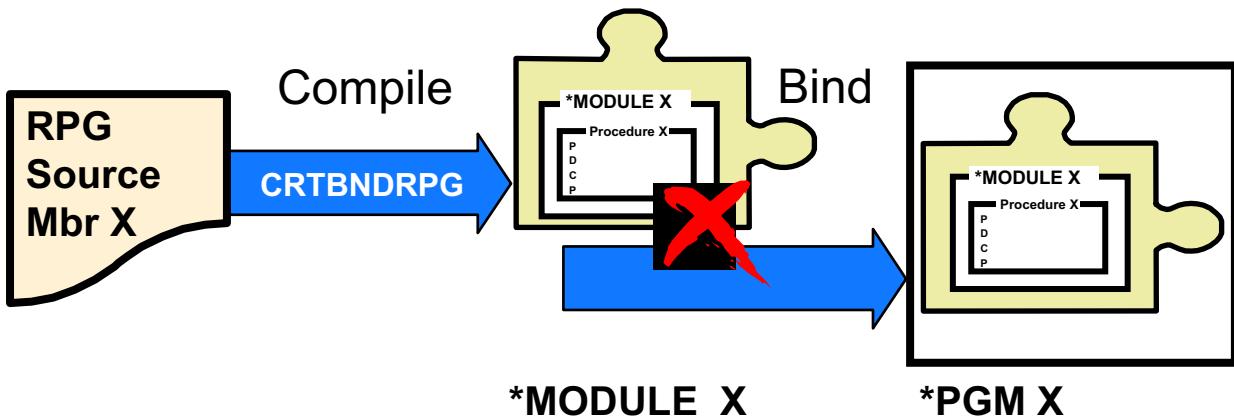
There are other ways to call programs that require less initiation time. Before we discuss these different calls, it is necessary to understand the terms used to describe objects and processes in ILE.

If you have a program that calls or is called by other programs, you must have created an executable object for each program before the application can be used.

## Creating non-ILE RPG IV programs (2 of 4)

IBM i

- ILE programs can contain one or more modules.
- For single module programs: CRTBNDRPG.
- Module deleted from QTEMP after Bind step.



© Copyright IBM Corporation 2009

Figure 10-5. Creating non-ILE RPG IV programs (2 of 4)

AS075.0

### Notes:

When you run CRTBNDRPG:

1. This command creates a \*MODULE object in QTEMP.
2. This command creates a \*PGM object that binds by copy the \*MODULE that was created in QTEMP.
3. The \*MODULE in QTEMP is automatically deleted at the end of the bind step, (whether successful or not).

Either the LPEX Editor's **Compile with Prompt** or Option 14 on the PDM Work with Members screen runs the CRTBNDRPG command for source member type RPGLE.

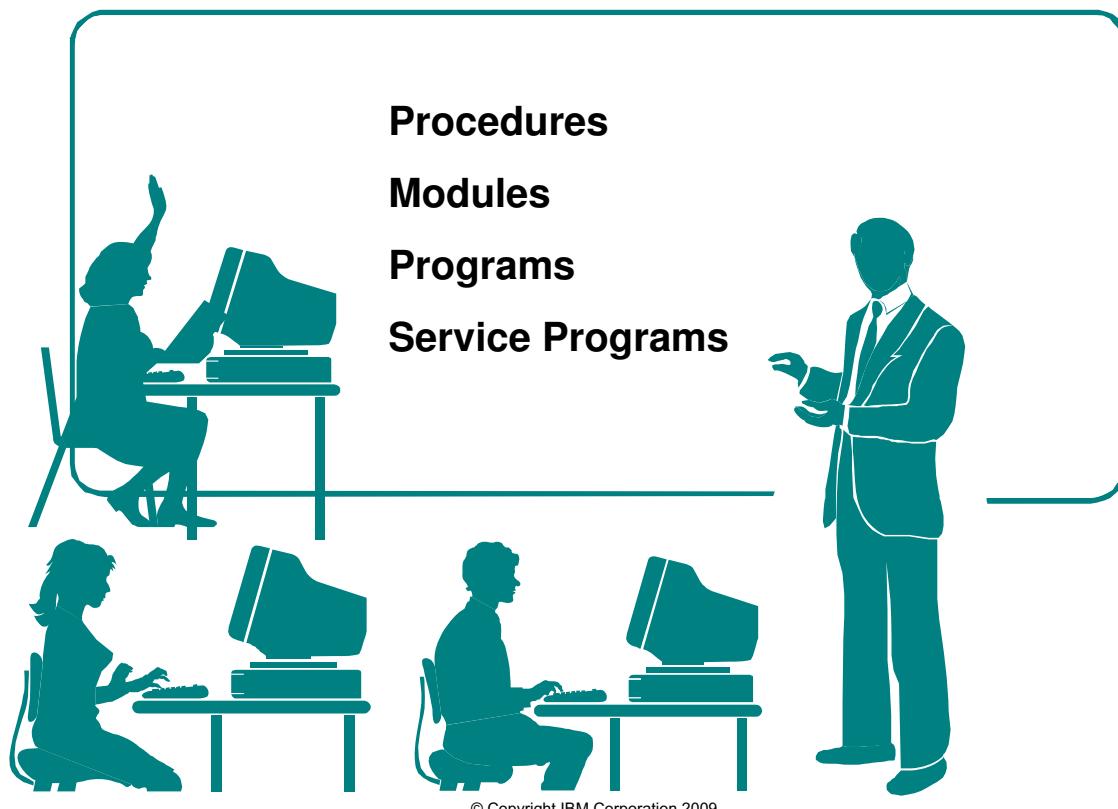
CRTBNDRPG provides a simple, *single-step* approach to program creation that provides very limited use of ILE facilities. There is a one-to-one relationship of:

**procedure:module:program**

The CRTBNDRPG command is equivalent to the old CRTRPGPGM command for the OPM RPG/400 language. We cannot take full advantage of ILE because the \*MODULE objects (our *building blocks*) are not retained.

# Definitions

IBM i



© Copyright IBM Corporation 2009

Figure 10-6. Definitions

AS075.0

## Notes:

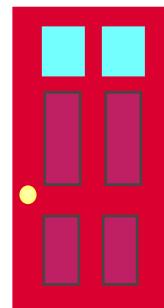
These are terms that are commonly used in ILE. New terminology and processes that we need to understand in order to be productive can often overwhelm us. We relate ILE terms to each other and where possible to a similar function that you perform when you write and maintain applications today.

We discuss each term very briefly.

# Procedure (1 of 2)

IBM i

- Entry point: Code that can be called:
  - Not an iSeries object
  - The code contained in the module/program objects
- Procedure specification varies by language
  - RPG IV, ILE C, and ILE COBOL
    - Many per compilation
    - COBOL uses nested programs
- ILE CL
  - One per compilation
- Call with CALL Procedure syntax:
  - Procedures appear on call stack
  - CALLP, CALLB, or Expression call in RPG IV
  - CALLPRC in CL



© Copyright IBM Corporation 2009

Figure 10-7. Procedure (1 of 2)

AS075.0

## Notes:

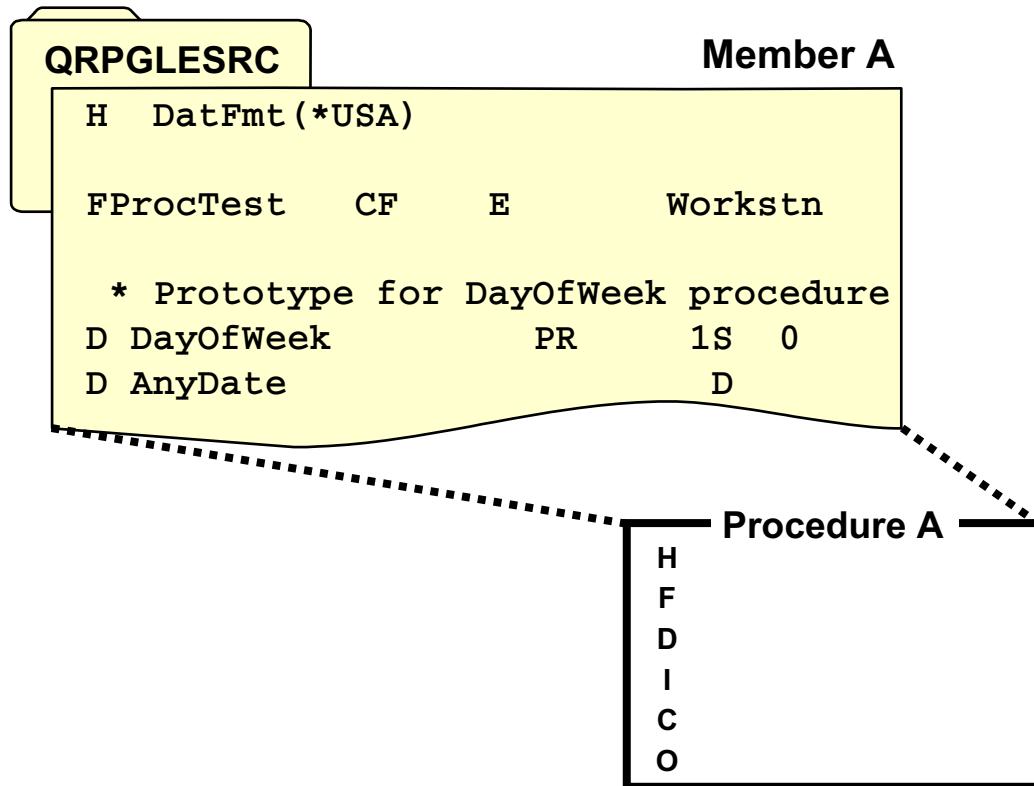
A procedure contains the entry point into the compiled code.

An entry point into code is something that can be called. In CL, there is only one external entry point per module currently. RPG IV can have multiple procedures per module. ILE COBOL can have multiple procedures when using the nested program support.

ILE C provides for many external entry points in a module (callable functions).

## Procedure (2 of 2)

IBM i



© Copyright IBM Corporation 2009

Figure 10-8. Procedure (2 of 2)

AS075.0

### Notes:

The *ILE Concepts* manual defines a *procedure* as:

"A set of self-contained high-level language statements that performs a particular task and then returns control to the caller."

A procedure can be written in any of the ILE supported languages, such as RPG IV, COBOL, and C.

The smallest executable component of code that you can write is a *procedure*.

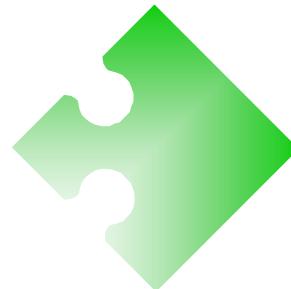
For RPG IV, you would code a source member of type **RPGLE**.

In the past, we have typically called this source member a *program*.

# Module (1 of 2)

IBM i

- \*MODULE object type:
  - Is created from a source member by CRTxxxMOD
  - Is the new object type for ILE
  - Houses the code for procedures that are not yet bound into programs
- It contains compiled, translated but not executable code.
  - Must be bound into a program to run.
- It can contain one or more procedures:
  - If written in a language that supports multiple procedures
- It can be deleted after code is bound into a program or service program.



© Copyright IBM Corporation 2009

Figure 10-9. Module (1 of 2)

AS075.0

## Notes:

The **module object** is created by a compile (**CRTxxxMOD**).

In general, modules are bound with other ILE modules to create a multi-module bound program object using the **CRTPGM** command.

The module object is simply a *container* for the code, which makes up the procedures, which are ultimately bound into one or more programs or service programs. You learn more about service programs later.

The code contained in the module object is completely compiled and translated.

Compilation and translation are two separate steps that make up what most programmers think of as *the compilation process*.

However, the code cannot be executed because there is no way to call the code, from a command line, for example, in the module. But binding the module into a program, or a service program, provides the necessary mechanism to enable you to call and execute the code.

## Module (2 of 2)

IBM i

### Member A

```
H DatFmt (*USA)
FProcTest CF E Workstn
* Prototype for DayOfWeek procedure
D DayOfWeek PR 1S 0
D AnyDate D
```

Procedure A

```
H
F
D
I
C
O
```

Compile

**CRTRPGMOD**

**\*MODULE A**

Procedure A

```
H
F
D
I
C
O
```

Figure 10-10. Module (2 of 2)

AS075.0

### Notes:

When you compile a procedure, you create a packaging container that is called a *module*, a **\*MODULE** object. Modules are not executable objects.

Modules must go through another step before they can be executed. A module is simply an intermediate *container* for compiled procedures. The module container provides a means of replicating object code in subsequent program and service program objects. A separate *bind* step, which we discuss later in the unit, achieves this replication.

## Program (1 of 2)

- \*PGM object types can now be either:
  - An OPM program (CRTxxxPGM), or
  - An ILE program (CRTPGM)
- ILE \*PGMs contain code from one or more \*MODULEs.
  - Still only one external entry point
- Both types are called with (standard) CALL program syntax.

© Copyright IBM Corporation 2009

Figure 10-11. Program (1 of 2)

AS075.0

### Notes:

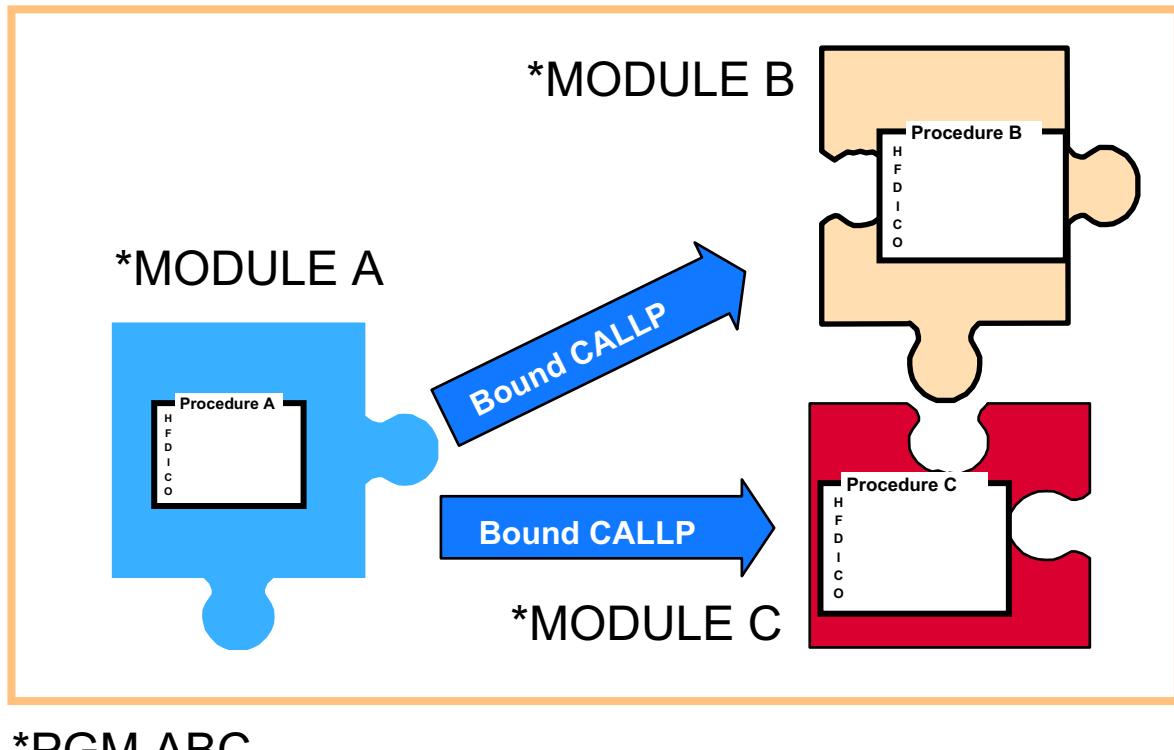
Programs are still called in exactly the same way they are in the OPM environment that you have been using. In some cases, their behavior is different if they are ILE programs.

ILE program objects are still called with the CALL syntax (from a command line, for example) used in OPM. If an ILE program contains multiple procedures, the procedures bound into that program are called using a bound (prototyped call recommended) call.

An ILE \*PGM module can contain modules created by different ILE compilers.

## Program (2 of 2)

IBM i



**\*PGM ABC**

© Copyright IBM Corporation 2009

Figure 10-12. Program (2 of 2)

AS075.0

### Notes:

A final step must be performed to create an executable **\*PGM** object. An ILE **\*PGM** contains one or more modules.

In ILE, we can create a program that can contain several RPG IV modules, plus a COBOL module, and a CL module. Here is where ILE differs from the traditional way of building i5(iSeries) programs. When we create an ILE program, it is similar to linking all the code together.

Why does this matter to you? Read the following paragraphs:

When you call one **\*PGM** from another using a Dynamic Call, the system has to locate the program, and open its resources. This involves some overhead and, if done often enough, can become a performance concern.

Except for the *first program* in an application, usually a call from the command line or a menu, you do not have to use the Dynamic Call in ILE. By using ILE create commands, you can group modules together that are *ready to run* contained in a **\*PGM** object. To pass control to another procedure within a program, and pass any parameters, we use

the ILE bound call to the \*MODULE that contains the procedure. RPG IV supports bound calls via the **CALLB** and **CALLP** opcodes as well as subprocedure calls from an expression. If you perform a bound call **from CL** to an RPG procedure, for example, you use the **CALLPRC** (Call Bound Procedure) command.

There is one more very important thing to note. Considering that all the resources needed are already allocated when the bound call is executed, the system can pass control to the called module more quickly.

When you create the program in this way, the modules are **Bound by Copy**. To create a program that binds modules by copy, you use the **CRTPGM** command.

# Service program (1 of 2)

IBM i

- It contains a collection of commonly used modules:
  - It can be thought of as a subroutine library.
  - One copy of module code is used (referenced) by many programs.
- Access via bound (fast) call:
  - It uses bind by reference.
  - \*SRVPGM object is never called.
  - Procedures contained in \*SRVPGM are called individually.
- Procedure in \*SRVPGM can be shared by different programs that call it.
- Code reuse and sharing are purposes of service programs.

© Copyright IBM Corporation 2009

Figure 10-13. Service program (1 of 2)

AS075.0

## Notes:

*Bind by copy* is one way to bind multiple ILE modules together to improve call performance at run time in a program. If a particular module is used in many programs, *Bind by reference*, using Service Program objects, is a better choice than bind by copy.

A service program is an ILE object that serves as a collection of modules (procedures) that are called using a *bound call* (also known as a *static call*) by program (\*PGM) objects.

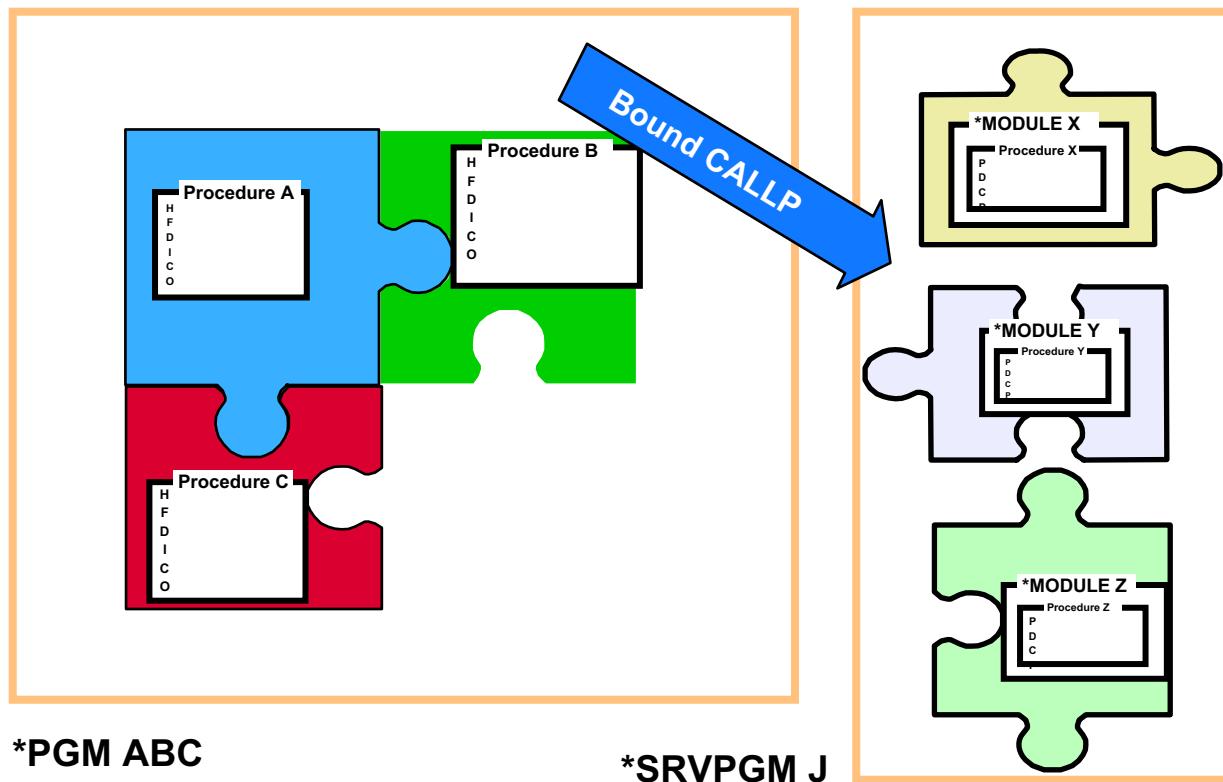
When a program is created that calls a procedure contained in a service program, the program is said to be *bound by reference* to the service program.

A Service Program object cannot be called as an individual object. It merely contains one or more procedures of code which are individually called.

Assume we have a Service Program named **UTIL** that contains modules B, C and D. The **UTIL** \*SRVPGM object cannot be called. However, bound calls (using a CALLB or CALLP opcode) can be issued to modules (procedures) B, C or D that are contained in **UTIL**.

## Service program (2 of 2)

IBM i



© Copyright IBM Corporation 2009

Figure 10-14. Service program (2 of 2)

AS075.0

### Notes:

In ILE, you can group commonly used procedures in ILE service programs. A **\*SRVPGM** is created by the command **CRTSRVPGM** (Create Service Program) and can be called by many different ILE procedures contained in **\*PGM** objects. A service program is not included in the **\*PGM** object as it is in the case of a Bind by Copy. A service program is an object external to the calling program. You use a *bound call* (CALLB, CALLP, or subprocedure call) to call the desired procedure.

When you access a **\*SRVPGM** in this way, it is called a *bind by reference*.

## Creating ILE RPG IV programs (3 of 4)

IBM i

- Edit: Code the procedure and create a member in QRPGLESRC
- Compile: Use CRTRPGMOD to create a \*MODULE object
- Repeat steps 1 and 2 for every module needed to create the ILE program
- Create program: Use ILE binding
  - One way is to bind modules in a \*SRVPGM
- Execute: Call program via command line, menu, and so forth

© Copyright IBM Corporation 2009

Figure 10-15. Creating ILE RPG IV programs (3 of 4)

AS075.0

### **Notes:**

This visual shows you the ILE process of creating and then executing an ILE program.

There are five phases now to create and run a program in the ILE world:

1. Coding: To use an editor to create a source member and enter or maintain source code.
2. Compilation: To use a language compiler to create object code that is *wrapped* in a \*MODULE container upon successful compilation.
3. Binding: To replicate and join \*MODULEs together into an executable load unit (\*PGM or \*SRVPGM)
4. Activation to load into main storage a executable unit (\*PGM) and complete any final bind processes (such as to a \*SRVPGM), allocate working storage, and so on.
5. Execution: To run the program, starting at the PEP (Entry Point), then initialize variables, and so on.

## Creating ILE RPG IV programs (4 of 4)

IBM i

- For multiple module programs: CRTRPGMOD + CRTPGM

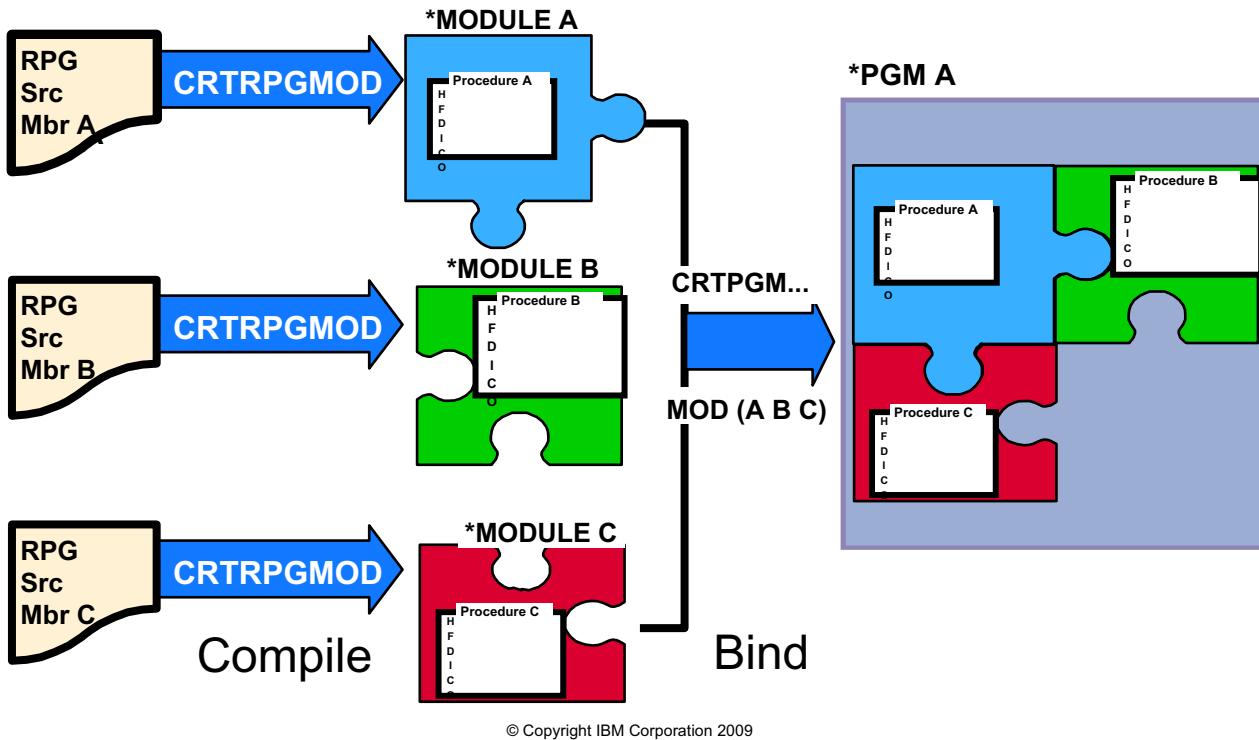


Figure 10-16. Creating ILE RPG IV programs (4 of 4)

AS075.0

### Notes:

This 2-step approach can also be used for creating a program that contains a single module. CRTBNDRPG simply provides a shortcut.

Option 15 from the PDM Work with Members screen starts the CRTRPGMOD command for member type RPGLE.

If you click **Compile with Prompt** in LPEX/RSE, you can select the CRTRPGMOD command for the compile.

# Creating ILE programs: Bind by copy

IBM i

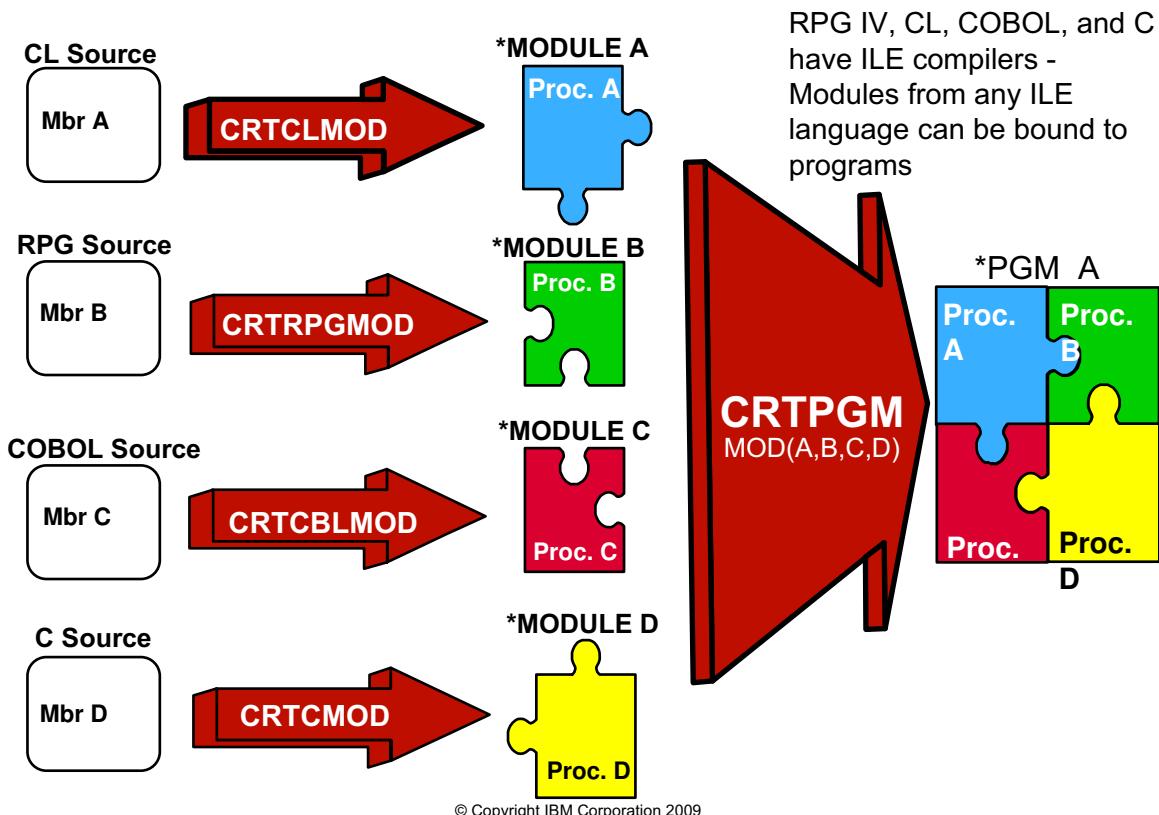


Figure 10-17. Creating ILE programs: Bind by copy

AS075.0

## Notes:

This visual illustrates the simplest form of static binding, bind by copy.

The *compile* step is the **CRTxxxMOD** step, which creates a **\*MODULE** object.

Each ILE language has its own syntax for calling bound procedures. Each of the source programs uses the bound call syntax specific to that language to call the procedures contained in the other modules of code.

To call a bound procedure from within a CL procedure, use the **CALLPRC** statement.

When all the module objects needed for a program are created, then the modules can be bound together using the **CRTPGM** command. In this example, all four modules are bound by copy into the program object, **A**. This is accomplished by specifying the four modules on the **MODULE** list parameter in the **CRTPGM** command:

- Module A
- Module B
- Module C
- Module D

As a result of the **CRTPGM** command, the program object now contains a **copy** of the compiled code of each of the four module objects. After completion of the **CRTPGM** command, the module objects may be deleted, if they are no longer needed in any other programs or service programs.

After program creation, if changes are needed to any specific module, the module can be re-created and that specific code replaced in the program by using the **UPDPGM** (update program) command. It is also possible to recreate a \*PGM using the **CRTPGM** command. However, it is not necessary to use it to replace a module in a bound program.

When the \*PGM is created, the system inserts the code necessary to make the \*PGM object executable. This code is the *PEP* or *Program Entry Procedure*. This is the entry point for an ILE program on a dynamic program call. Once again, the *UEP* or *User Entry Procedure* is the point in your procedure where execution begins. The UEP is in the procedure within the module that is given control when the PEP code has been run. A UEP is the entry point for your code in a program.

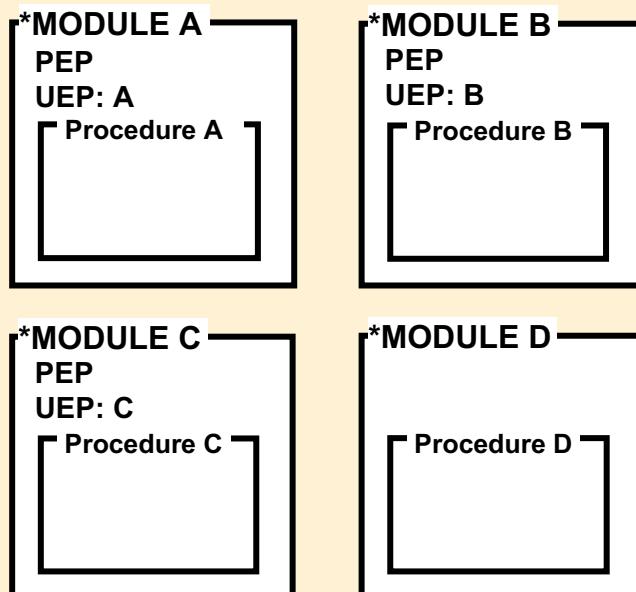
In a \*PGM, many modules have UEPs. You specify which one is the UEP for your \*PGM object in the ENTMOD parameter when you run **CRTPGM**.

# PEP and UEP

IBM i

## \*PGM A

### Program Entry Procedure: Use PEP in Module A User Entry Procedure: Use Procedure A in Module A



© Copyright IBM Corporation 2009

Figure 10-18. PEP and UEP

AS075.0

### Notes:

We now understand that a program can contain modules that have been created by any ILE language.

For each \*MODULE object, the system establishes an entry point. This is called a *Program Entry Procedure* and the system adds some code to initialize the module. A program entry procedure is the compiler-generated code that is the *entry point for an ILE program on a dynamic program call*. It is similar to the entry point in an OPM program. When you create a \*PGM object, the system establishes a PEP for the \*PGM object. This process was always done in the past. We just never realized it. When you dynamically call a \*PGM object, the PEP for the \*PGM is placed in the Call Stack.

Every procedure that you write has a point where execution begins, for example, your \*INZSR routine. The point where the code of each procedure you wrote begins execution is called the *User Entry Procedure*. A user entry procedure, written by a programmer, is the target of the dynamic program call. It is the procedure that gets control from the PEP. In other words, the procedure's code associated with the program entry procedure is the user entry procedure.

When you use the CRTPGM command to create the \*PGM object, you specify the point where you want control to be passed to begin execution of your code. CRTPGM will default to the first module in the list of modules to be bound. You can specifically code the name of the module that contains the UEP.

In our example, Modules A, B, and C contain a PEP. Any of these three could be designated as the Entry Module for the program. Module D has been created with no PEP. It cannot be used to supply the PEP or UEP for the program. Procedure D can be called only by another procedure within the program.

# CRTPGM command and ENTMOD

IBM i

## Create Program (CRTPGM)

Type choices, press Enter.

```
Program . . . . . . . . . . . . PGM      > A
  Library . . . . . . . . . . . .          > MYLIB
Module . . . . . . . . . . . . MODULE     > A
  Library . . . . . . . . . . . .          > MYLIB
                                + for more values > B
                                                > MYLIB
                                                > C
                                                > MYLIB
                                                > D
                                                > MYLIB
Text 'description' . . . . . . . . . . . *ENTMODTXT
```

## Additional Parameters

```
Program entry procedure module    ENTMOD      > A
  Library . . . . . . . . . . . .          > MYLIB
```

### Question

Which module provides the Entry Point for program A?

© Copyright IBM Corporation 2009

Figure 10-19. CRTPGM command and ENTMOD

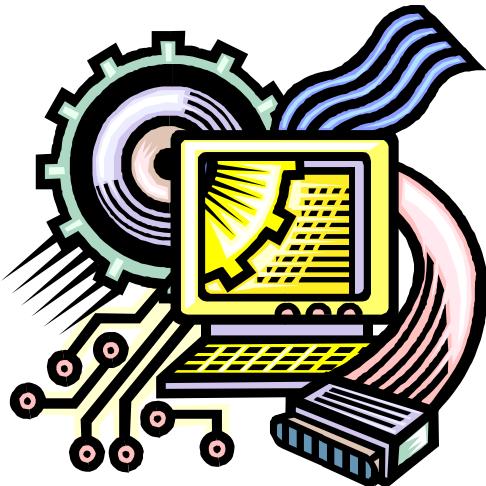
AS075.0

### Notes:

This visual shows the CRTPGM command specifying the Program Entry Point Procedure parameter.

# Machine exercises: Creating ILE objects and bind by copy

IBM i



© Copyright IBM Corporation 2009

Figure 10-20. Machine exercises: Creating ILE objects and bind by copy

AS075.0

## Notes:

Perform the machine exercises:

- Creating ILE objects
- Bind by copy

## 10.3. Service programs

# Why service programs?

Bind by copy means:

- Many copies of frequently used routines
- Maintenance can be more complex

Solution is service programs (\*SRVPGM):

- Similar to subroutine library
- Single copy of frequently used routines
- Call performance similar to bind by copy:
  - Same bound procedure call
  - Additional overhead of initial activation; completing the bind

© Copyright IBM Corporation 2009

Figure 10-21. Why service programs?

AS075.0

## Notes:

The limitations of bind by copy might be obvious. If the same module is used in many programs, then you have the problems shown here. The solution to these problems is to put commonly used modules into a service program, rather than binding them by copy into the \*PGM object.

Using service programs, the calls are still bound procedure calls, using the bound call syntax and achieving the same bound call performance. But only one copy of the module code is needed. The module is contained in a service program that can be used by many different programs.

Service programs are similar to a DLL (dynamic link library) in a PC programming environment.

# Creating a service program

IBM i

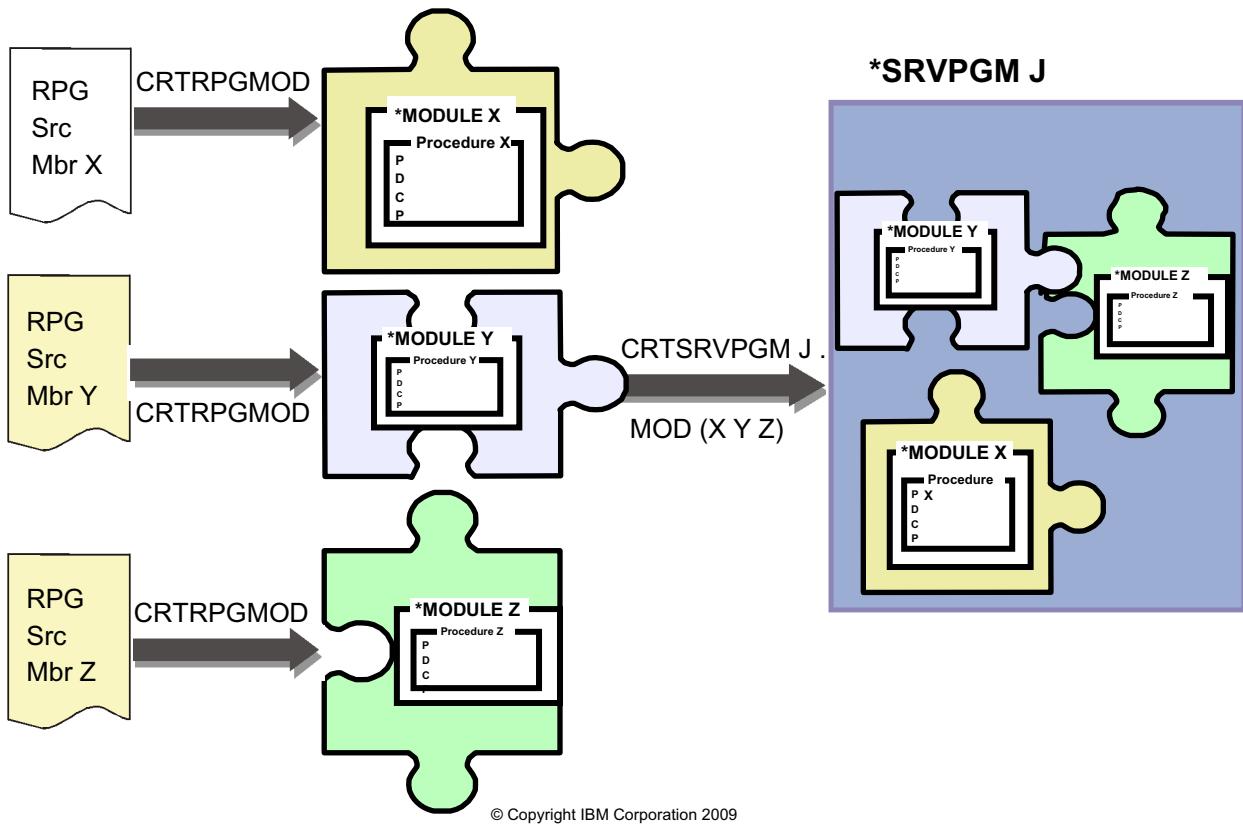


Figure 10-22. Creating a service program

AS075.0

## Notes:

This visual illustrates creating a service program object.

This process is very similar to that shown in the CRTPGM visual where we created a program object with bind by copy. Although programs and service programs are created in a similar fashion, how they are used is very different.

Service Program objects are never called. They are referenced by \*PGM objects. The procedures contained inside a referenced Service Program object, however, can be called by ILE \*PGM objects, using a bound call or a procedure call.

If a module contained in a service program changes, the old module can be replaced with the new module using the UPDSRVPGM (Update Service Program) command.

# CRTSRVPGM command

IBM i

## Create Service Program (CRTSRVPGM)

Type choices, press Enter.

Service program . . . . .	SRVPGM	> J
Library . . . . .		> MYLIB
Module . . . . .	MODULE	> X
Library . . . . .		> MYLIB
		> Y
		> MYLIB
	+ for more values	> Z
		> MYLIB
Export . . . . .	EXPORT	*SRCFILE
Export source file . . . . .	SRCFILE	QSRVSRC
Library . . . . .		*LIBL
Export source member . . . . .	SRCMBR	*SRVPGM
Text 'description' . . . . .	TEXT	*BLANK

© Copyright IBM Corporation 2009

Figure 10-23. CRTSRVPGM command

AS075.0

## Notes:

This visual shows the CRTSRVPGM command.

# Using service programs

IBM i

- Bind by copy and bind by reference can be used in same program.
- Completion of bind between Program A and Service Program J occurs at call time of Program A.

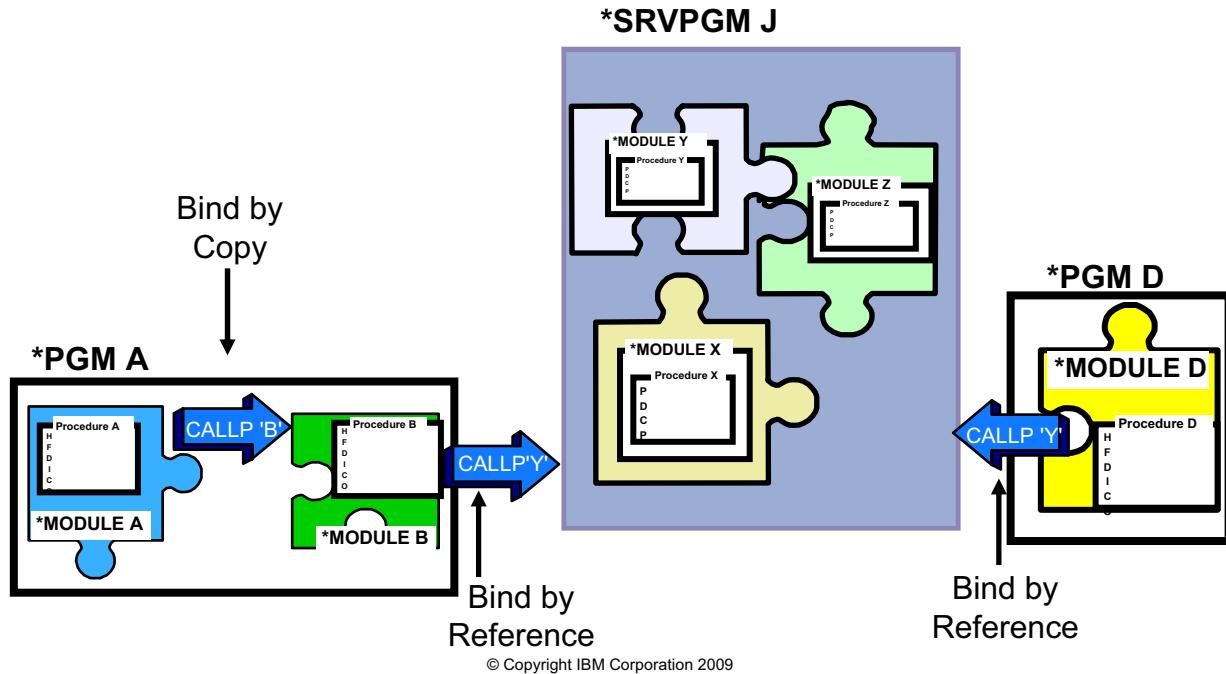


Figure 10-24. Using service programs

AS075.0

## Notes:

This visual illustrates creating a program that references the Service Program named J.

Notice that at least one module must be listed in the Module list of the CRTPGM command and this module will be bound into the program object by copy (that is, its code will be copied into the program object).

The bind to service program J is completed at run time. Performance of this run-time bind is similar to a dynamic call (also known as a dynamic bind). Whereas dynamic binding would occur in every dynamic call to a traditional program, the linkage between \*PGM A and \*SRVPGM J occurs once when \*PGM A is called. From that point, any and all calls to **Procedure (module) Z** perform like the static (bound) call. This is true for all other procedures stored in service program J.

When creating service programs, you should plan what code is contained within it. A suggestion is to group procedures that are application or function related in the same service program. For example, all payroll routines would be placed in a single \*SRVPGM while all date routines would be in another \*SRVPGM.

# CRTPGM command and BNDSRVPGM

IBM i

```
Create Program (CRTPGM)

Type choices, press Enter.

Program . . . . . . . . . . . . . PGM      > A
  Library . . . . . . . . . . . . .          > MYLIB
Module . . . . . . . . . . . . . MODULE     > A
  Library . . . . . . . . . . . . .          > MYLIB
                                         + for more values
                                         *LIBL
Text 'description' . . . . . . . TEXT       *ENTMODTXT

Additional Parameters

Program entry procedure module   ENTMOD      *FIRST
  Library . . . . . . . . . . . . .          > J
Bind service program : . . . . . BNDSRVPGM
  Library . . . . . . . . . . . . .          > MYLIB
                                         + for more values
                                         *LIBL
```

© Copyright IBM Corporation 2009

Figure 10-25. CRTPGM command and BNDSRVPGM

AS075.0

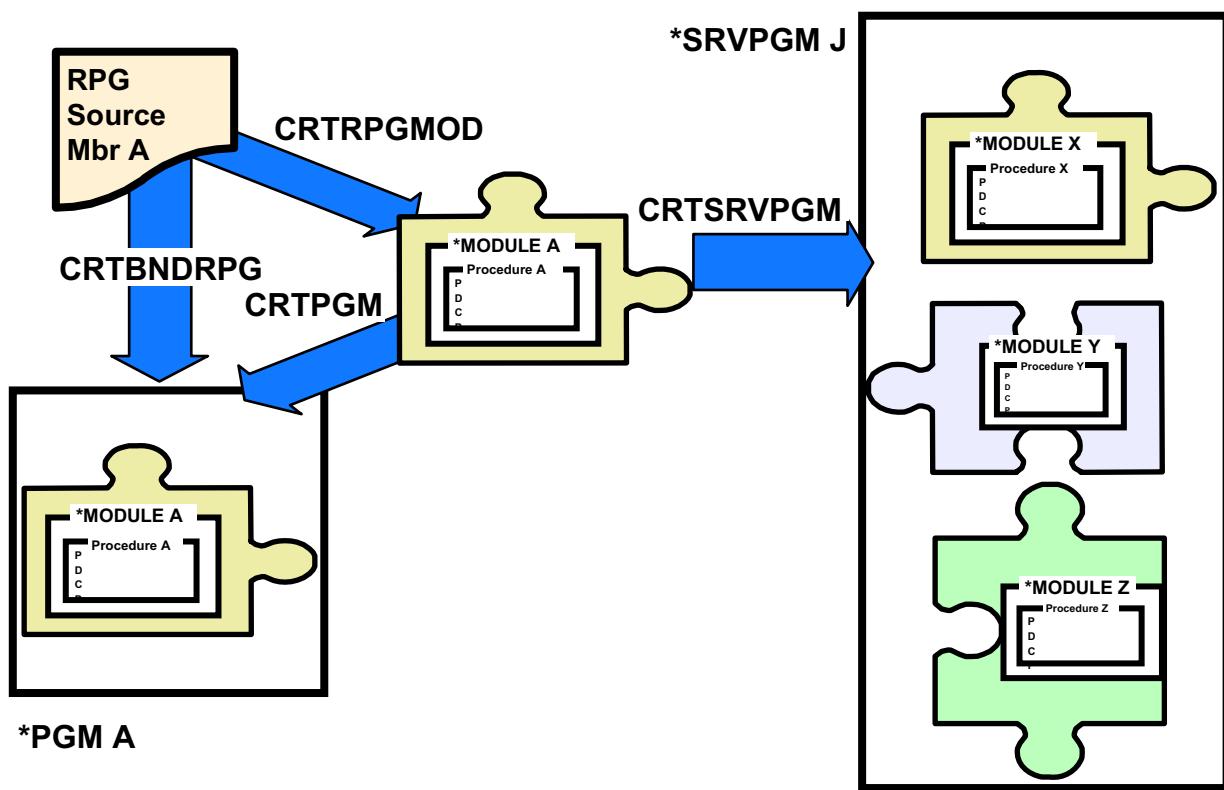
## Notes:

When you create an ILE program using bind by reference, you must always include one module to be bound by copy. You can specify more than one module as you can mix bind by copy and bind by reference.

On the lower part of the visual, notice the BNDSRVPGM parameter. This binds the service program to your program object and enables you to bind to the modules in the service program by reference.

# Creating ILE objects

IBM i



© Copyright IBM Corporation 2009

Figure 10-26. Creating ILE objects

AS075.0

## Notes:

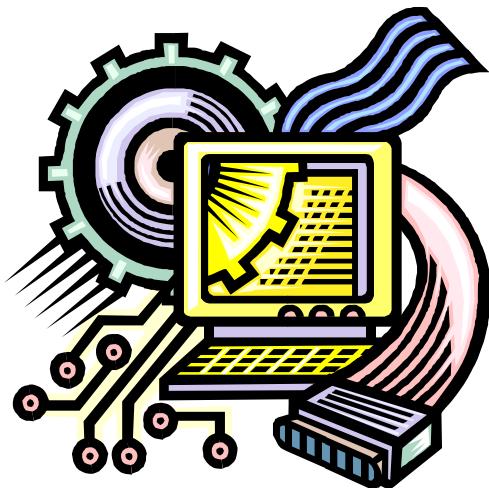
This visual summarizes the different ways that you can create ILE objects by using different commands.

We have the choice of where we package the Module A. We could place it in a Service Program. We have chosen to bind it into a program.

ILE gives us these packaging choices.

## Machine exercises: Bind by reference

IBM i



© Copyright IBM Corporation 2009

Figure 10-27. Machine exercises: Bind by reference

AS075.0

### Notes:

Perform the machine exercise “Bind by reference.”

## 10.4. Call types

# Modular programming

IBM i

- To write modular code means:
  - Reuse of proven code
  - More flexibility in workload distribution
  - Smaller modules of code can be in production sooner
  - Ability to purchase and use commercially available routines
  - Single-function code easier to maintain
  - Applications more easily adapted for changing business needs
- Dynamic call sometimes means:
  - Performance cost
  - Large, multifunction programs; more difficult to maintain
- ILE static binding:
  - Reduces performance price of frequently called routines
  - Encourages modular application design

© Copyright IBM Corporation 2009

Figure 10-28. Modular programming

AS075.0

## Notes:

Although you can create RPG IV programs in the traditional fashion, the RPG IV compiler offers additional function when you take advantage of the Integrated Language Environment.

One of the most important features of ILE is the ability to do static binding of program modules. Static binds provide faster call times between individually written and compiled parts of an application. Modules are actually combined prior to runtime to produce a program object.

This visual lists some of the advantages of being able to write modular applications and yet achieve optimum performance when calling another module at run time. This is what ILE's static binding capability enables.

# Dynamic CALL versus static CALL

IBM i

- Dynamic Program Call:
  - Legacy HLL CALL opcode in RPG IV (also COBOL and CL)
  - In RPG IV, CALLP is a better option
- Static (bound) Procedure Call:
  - CALLP or function call for RPG IV
  - CALLPRC for CL
  - CALL LINKAGE PROCEDURE for COBOL
  - Function call in C
  - Calls a procedure
    - Bound by copy or reference

© Copyright IBM Corporation 2009

Figure 10-29. Dynamic CALL versus static CALL

AS075.0

## Notes:

Two call operations are available to you in ILE:

- Call bound procedure
- Call program

The syntax for the call bound procedure operations is shown in the visual for various languages.

For your information, the COBOL compiler has some additional options to allow programmers to use the *normal* CALL statement in the procedure division, and to specify either in special names or as a compile option which type of CALL is the default for this program or for each program or procedure name called.

Why is the Bound (Static) Call so useful? Because the Bound Call performs better than the Dynamic Call we can use it more widely in our application without impacting performance. This means we can build code components that are much more modular in design. As we have already seen, modular coding is very important for improving maintainability and the reuse of code.

# Example: Dynamic CALL versus static CALL

IBM i

```

FIG1017B.RPGLE X
Line 29      Column 1      Insert
. .... / . 1 .....+.... 2 .....+.... 3 .....+.... 4 .....+.... 5 .....+.... 6 .....+.... 7 .....+.... 8 .....+.... 9.

000100
000200      // Prototypes for Program and Procedure call operations
000300
000400      D DayOfWeekPgm    PR          ExtPgm( 'DAYOFWEEK' )
000500      D   AnyDate
000600      D   DayNum        1S 0
000700
000800      D DayOfWeekProc1  PR          D
000900      D   AnyDate
001000      D   DayNum        1S 0
001100
001200      D DayOfWeekProc2  PR          1S 0
001300      D   AnyDate
001400
001500      D InputDate     S          D   Inz(*Job)
001600      D DayNumber     S          1S 0
001700
001800
001801
001900      /Free
002000
002100      CallP DayOfWeekPgm(InputDate:DayNumber); // Dynamic Program Call
002200
002300      CallP DayOfWeekProc1(InputDate:DayNumber); // Static Procedure Call
002400
002500      DayNumber = DayOfWeekProc2(InputDate); // Static Procedure Call (expression)
002600
002700      *InLR = *On;
002701
002800      /End-Free

```

© Copyright IBM Corporation 2009

Figure 10-30. Example: Dynamic CALL versus static CALL

AS075.0

## Notes:

In the past, we might have used the RPG IV fixed-format **CALL** opcode to perform a Dynamic Call to another program.

The equivalent fixed-format **CALLB** (or a bound call) is a Static Call to another procedure. It is more logical to think of these calls in this way:

- CALL = Program Call
- CALLB (Bound CALLP) = Procedure Call

**Note:** A Procedure Call targets a different object than a Program Call!

With the (semi) free-format syntax of RPG IV, we can use the CALLP operation as a multipurpose, prototyped call that can be used to execute either programs or procedures (that is, it can be used as a CALL or a CALLB).

Procedures can also be executed within an expression, often referred to as a function call. This is very similar to the use of built-in functions.

# Static binding

IBM i

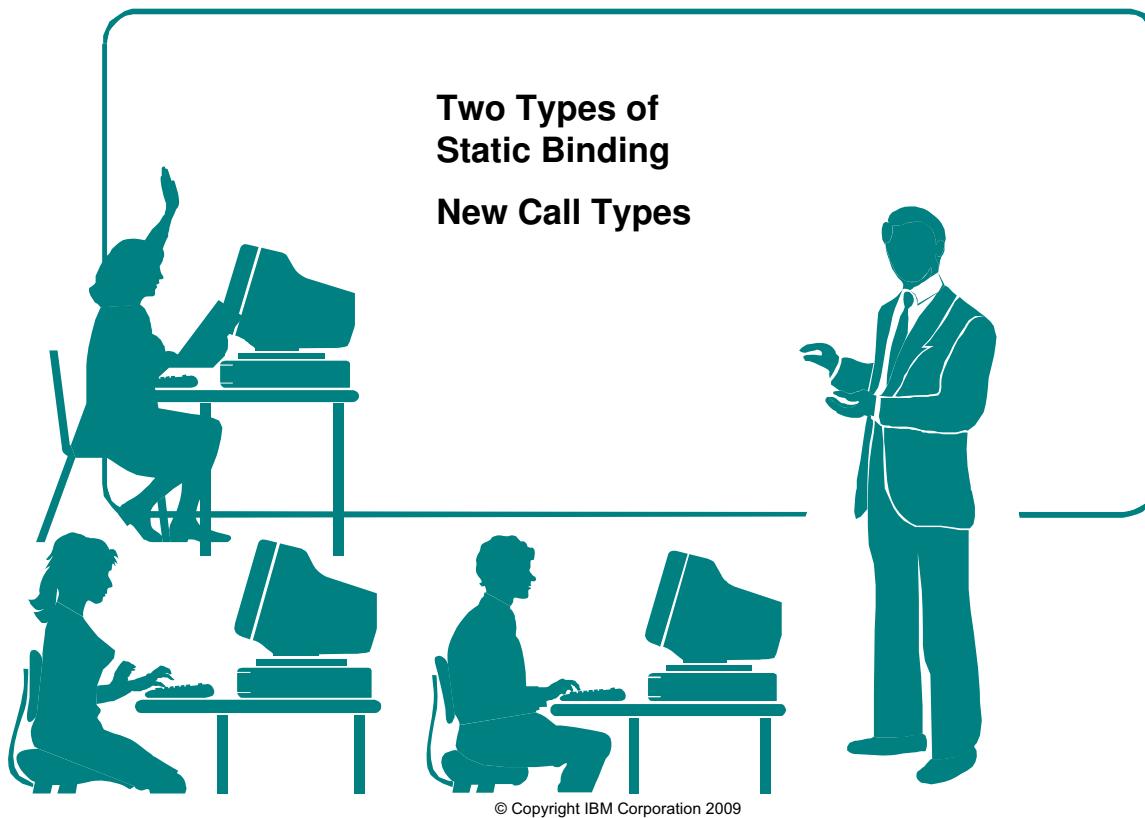


Figure 10-31. Static binding

AS075.0

## Notes:

The term *binding* might require some clarification. When you used the dynamic call in the prototyping exercise, you performed a *dynamic call*. When the CALLP was executed, the called program was bound to the calling program dynamically. Hence, *dynamic binding* was performed by the CALL.

On the other hand, ILE performs a *static call*. ILE calls are static in nature because the binding is performed at the time the program object is created.

So, there are two types of calls: static and dynamic. The type of binding performed determines the type of call you use. Binding happens on all calls depending on when the bind is resolved - at call time (dynamic) or at program creation time (static).

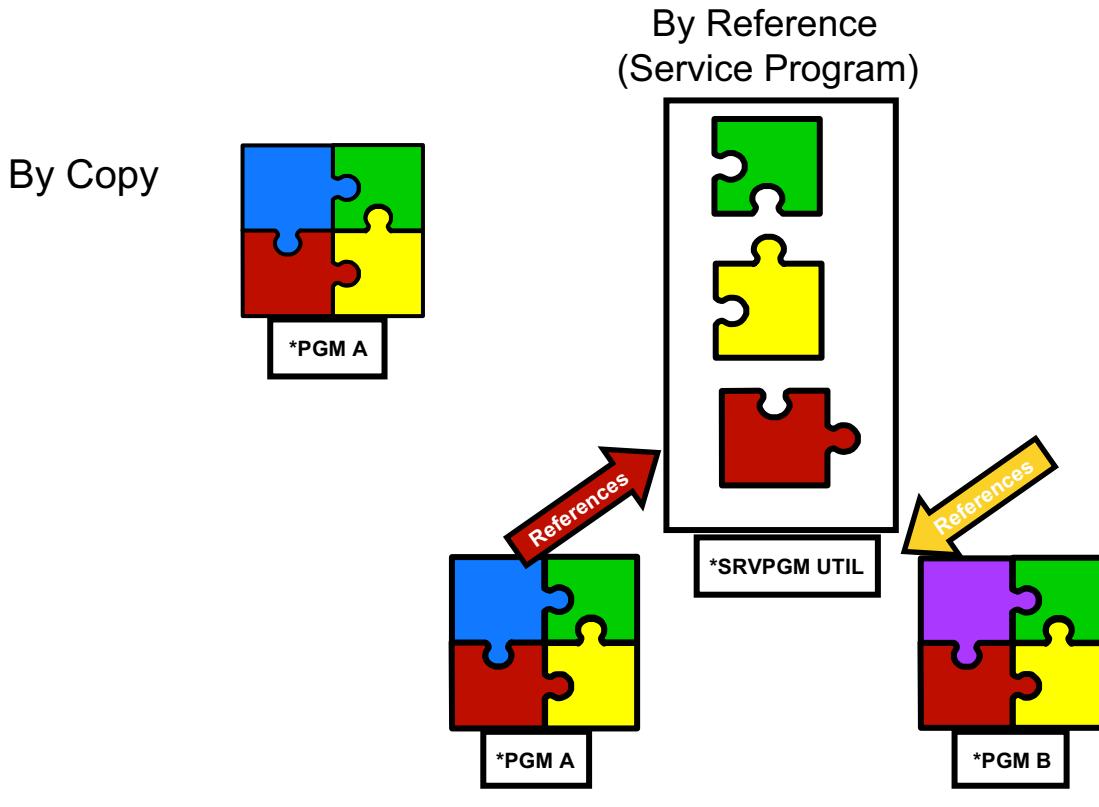
In the world of ILE, we need to be specific in describing the type of bind that is being performed. To simply use the term *bind* is no longer sufficient.

Static binding is discussed in the *ILE Concepts* manual. The *ILE RPG Programmers Guide* also uses this term. Also, the help text for CRTBNDRPG also uses the term *static binding* to define the meaning of DFTACTGRP(\*NO).

The benefit of binding is that it helps reduce the overhead associated with calling programs. Binding the modules together speeds up the call. The legacy call mechanism is still available, but there is also a faster alternative. To differentiate between the two types of calls, the previous method is referred to as a *dynamic* or external program call, and the ILE method is referred to as a *static* or *bound procedure call*.

# Bind by copy and bind by reference

IBM i



© Copyright IBM Corporation 2009

Figure 10-32. Bind by copy and bind by reference

AS075.0

## Notes:

ILE offers two types of static binding:

1. *Bind by copy* is a form of static binding where the compiled code is copied from the \*MODULE object, created by the compilation step, into the program. This happens at the bind step, which is initiated with the CRTPGM command. We look into this process later in this topic.
2. *Bind by reference* is a type of binding where the program refers to one or more procedures in a Service program. The complete Service program is not physically copied into the program, but is *logically* bound to the program. The term *bind by reference* is only associated with Service programs. A Service program serves as a container for a set of procedures that are used (called using a Bound Call statement) by program (\*PGM) objects.

Multiple programs can be bound by reference to the same service program. This allows you to reuse your code without having to create many copies of compiled code in the resulting programs.

Note that in addition to the ILE facility of static binding, dynamic (program call) binding still exists and can be used in ILE applications.

*Resolution of Procedure Calls* can occur completely at bind time (bind by copy) or partially at bind time and finally at activation time (bind by reference).

**Key Points:**

- There is *no* binding performed at compile time.
- There is *no* resolution of Procedure Calls at execution (run) time.
- Only Program Calls (dynamic or late bind) are resolved at execution time.

# Why two types of static binding?

IBM i

	Bind by Copy	Bind by Reference
Speed of Call	Fast	Fast
Structural Complexity	Simple	More complex
Best suited for calls	From one program	From many programs

- What about Dynamic Binding (calls to \*PGMs)?
  - Dynamic calls to programs are still supported.
  - This is a good choice for code not called frequently.
  - It is activated in the job only if and when called.
    - Statically bound modules are always activated together.

© Copyright IBM Corporation 2009

Figure 10-33. Why two types of static binding?

AS075.0

## Notes:

ILE's binding capability, together with the resulting improvement in call performance, makes it far more practical to develop applications in a highly modular fashion. An ILE compiler does not produce a program that can be run. Rather, it produces a module object (\*MODULE) that can be combined (bound) with other modules to form a single runnable unit; that is, a program object (\*PGM).

Just as you can dynamically call an RPG program from a COBOL program, ILE allows you to *bind modules written in different languages*. Therefore, it is possible to create a single runnable program that consists of modules written separately in the ILE languages: RPG, COBOL, C, and CL.

A static procedure call transfers control to an ILE procedure. Static procedure calls can be coded only in ILE languages. A static procedure call can be used to call any of the following:

- A procedure within the same module
- A procedure in a separate module within the same ILE program or service program
- A procedure in a separate ILE service program

## **Static bind by copy**

When using only bind by copy, all the code needed to run the function is located in the \*PGM object. This provides a simple application structure with no extra object inter-relationships (for example, to Service Programs).

On the other hand, when a module of code is called by many different programs, Bind by Copy results in the same module being copied into many program objects. This takes up space (on disk and at run time in memory). When one of those modules changes, each program using that module must be updated or rebound to get the new function, making maintenance more difficult.

## **Static bind by reference**

Bind by reference provides the same fast call support as bind by copy. However, there is now a minimum of one extra object involved (one or more Service Programs); so the application's structure becomes somewhat more complex. When a module of code is used by many programs, the Service Program provides support for a single shared copy of that module which simplifies maintenance and reduces the overall size of the application. Thus, using Service Programs and Bind by Reference is a good choice when the module of code will be called from many places (programs), but not a good idea when the module of code will be called from only one program.

## **Dynamic binding**

Dynamic binding, the type of binding done between \*PGM objects, in ILE as well as in OPM, is sometimes the best choice. When code is called infrequently, you do not want it activated on every use of a program. It is best to leave infrequently called code as separate programs to be dynamically bound only when needed.

# Updating a \*PGM

IBM i

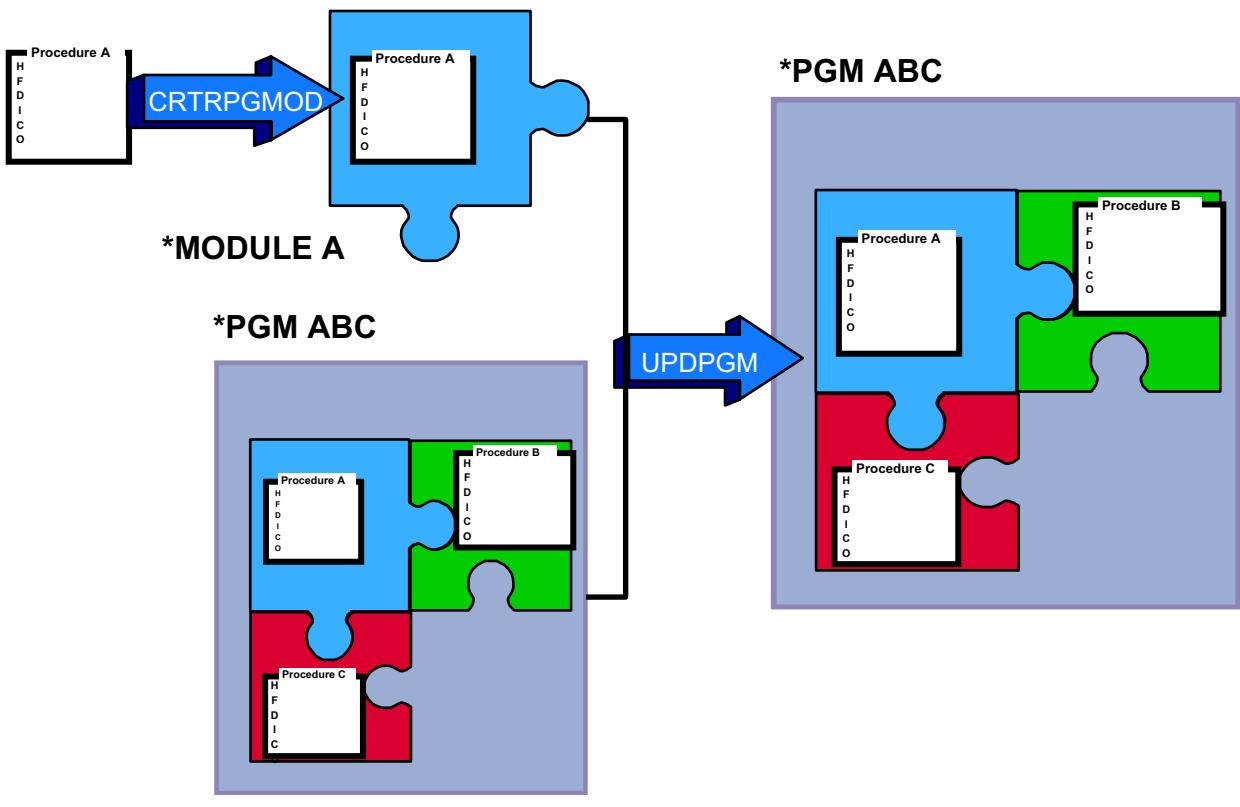


Figure 10-34. Updating a \*PGM

AS075.0

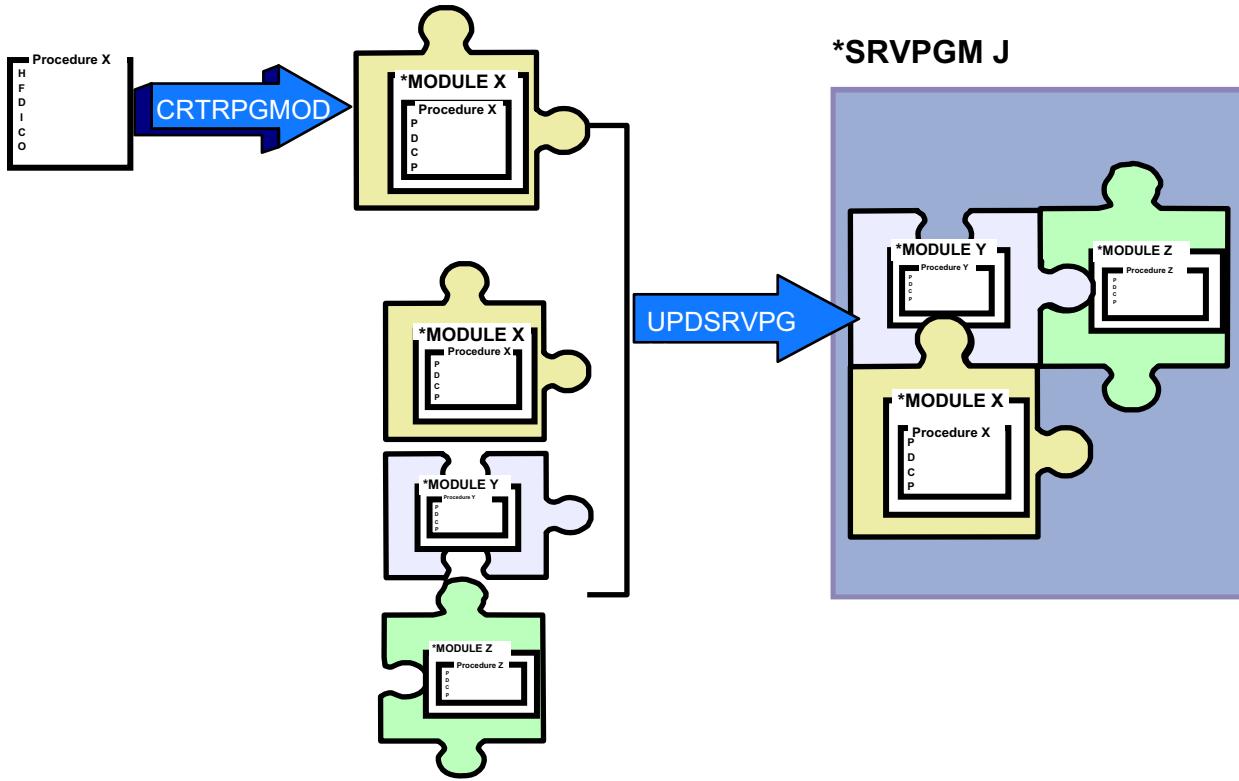
## Notes:

If you update a procedure that is bound by copy, you must first recreate the module using **CRTRPGMOD** command. Then you can rebind the changed module to all \*PGMs that require it using the **UPDPGM** command.

Of course, you can use **CRTPGM** to do the same thing. The difference is that the **UPDPGM** command requires only the name of the \*PGM and the modules that are to be replaced.

# Updating a \*SRVPGM

IBM i



© Copyright IBM Corporation 2009

Figure 10-35. Updating a \*SRVPGM

AS075.0

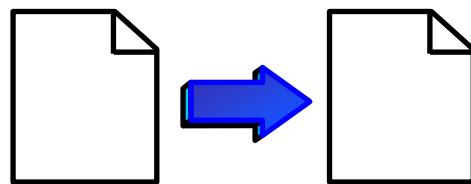
## Notes:

Like the UPDPGM command, **UPDSRVPGM** takes a changed module and replaces the existing module to create a new copy of the service program.

# Sharing data in ILE programs

IBM i

- Modules bound together can share data items:
  - One module defines and exports data.
  - One or more modules bound to the exporter can import the data.
- An alternative to other methods of sharing data:
  - More convenient than passing parameters
  - Safer than using LDA



© Copyright IBM Corporation 2009

Figure 10-36. Sharing data in ILE programs

AS075.0

## Notes:

Modules sharing data must be bound together, that is, they must use a *bound call*. The EXPORT and IMPORT keywords are used to define shared data.

**Export:** An export is the name of a procedure or data item, coded in a module object, that is available for use by other ILE objects. The export is identified by its name and its associated type, either procedure or data.

An export is also called a *definition*.

**Import:** An import is the use of or reference to the name of a procedure or data item not defined in the current module object. The import is identified by its name and its associated type, either procedure or data.

An import is also called a *reference*.

Passing parameters can become cumbersome in long invocation stacks. For example, if module D needs data from module A, but the invocation stack is A calls B calls C calls D, the parameter must be passed through all the intermediate modules, even though they do

not use it. Using import and export, module A can define the data and export it. Module D can then import that data item, with no impact on the calls to B or C.

# IMPORT and EXPORT keywords

IBM i

- Shared data defined on the D-spec with keywords:
  - EXPORT: This allows data to be used by another module.
  - IMPORT: This data is stored in the exporter module.
- Allowed for data structures and stand-alone fields and arrays.
- Exported data is initialized *only* when \*PGM containing the module is called.
  - Not reinitialized after LR in the exporting module.
- Multiple modules can IMPORT a specific data item.
- Only one module can EXPORT a specific data item.

© Copyright IBM Corporation 2009

Figure 10-37. IMPORT and EXPORT keywords

AS075.0

## Notes:

With data, the EXPORT and IMPORT keywords are specified on the D-spec in the keywords area for the data item.

Imported and exported data structures must be named.

Compile-time or pre-runtime arrays or data area fields cannot be imported.

Exported data items are handled as part of the program object (not procedure or module) where they are used. So they are initialized when the program that contains them is first called. The value of an exported data item is not initialized again, regardless of the status of LR when leaving either the procedure that exported it or the procedures that import it. This behavior is significantly different from fields *local* to the procedure or module.

# IMPORT and EXPORT example

Module A

D	ShardArray	S	5	DIM(10) EXPORT
---	------------	---	---	----------------

Module D

D	ShardArray	S	5	DIM(10) IMPORT
---	------------	---	---	----------------

- 1 Procedure A issues Bound CALLP to Procedure B
- 2 Procedure B issues Bound CALLP to Procedure C
- 3 Procedure C issues Bound CALLP to Procedure D

© Copyright IBM Corporation 2009

Figure 10-38. IMPORT / EXPORT example

AS075.0

## Notes:

**ShardArray** is not defined in procedure B nor in procedure C. Because procedure D has an import request for **ShardArray**, the address in Procedure A is made available. Neither Procedure B nor procedure C know about **ShardArray**.

Each procedure (A and D) has its *own* copy of the index into **ShardArray**.

Export/Import is similar to passing a parameter.

## ILE is much, much more

IBM i

- Multiple procedure module
- Packaging subprocedures in NOMAIN procedure
- EXPORT for procedures and subprocedures
- Binding directories and binder language
- Activation groups
- Signatures
- ILE error handling

© Copyright IBM Corporation 2009

Figure 10-39. ILE is much, much more

AS075.0

### Notes:

You have made a good start in understanding how to use the feature of ILE. But, there is much more work to do:

- We have dealt with simple modules. We need to cover the more complex situation involving creating programs that reference many modules.
- There are different options for packaging procedures. The NOMAIN procedure is a very efficient packaging tool that improves performance by eliminating the cycle overhead built into every procedure by the RPG compiler.
- The EXPORT keyword can also be used for procedures. It enables you determine what code should be made available outside a main procedure.
- Activation groups allow better control of things like commitment control and must be used carefully.
- ILE has a signature attached to modules. You can have multiple copies of the same procedure in a service program. You can determine which one is used by signature. This is a great way to phase in a new release over time.

- ILE offers new and different error handling. Errors are not handled in the way to which we are accustomed.

# Unit summary

IBM i

Having completed this unit, you should be able to:

- Describe the benefits of ILE
- Create an ILE module
- Create an ILE service program
- Create an ILE program using static binding

© Copyright IBM Corporation 2009

Figure 10-40. Unit summary

AS075.0

## Notes:



# Unit 11. What next?

## What this unit is about

This unit describes the activities that students can perform to apply the knowledge gained in this class and to prepare to attend the follow-on class:

*AS10/S6199: i5(iSeries) RPG IV Version 5 Programming  
Advanced Workshop*

This is the second class in a series of three classes designed to help you to become a skilled RPG IV programmer. This unit discusses what is taught in the follow-on course and why it is important for you to attend.

## What you should be able to do

After completing this unit, you should be able to:

- List the prerequisite skills needed prior to attending the Advanced RPG IV classes
- List the topics of the next class

# Unit objectives

IBM i

After completing this unit, you should be able to:

- List the prerequisite skills needed prior to attending the Advanced RPG IV classes
- List the topics of the next class

© Copyright IBM Corporation 2009

---

Figure 11-1. Unit objectives

AS075.0

## **Notes:**

## What have you learned?

IBM i

- How to use OVERLAY and related DDS keywords to develop efficient interactive programs
- How to code interactive programs that support inquiry of subfiles
- How to code interactive programs that support maintenance of subfile records
- How to use arrays and data structures in RPG IV programs
- How to anticipate and manage common errors and exceptions
- How to write prototyped calls
- How to write and call RPG IV subprocedures
- How to create ILE modules and include those modules in program objects using bind by copy and bind by reference

© Copyright IBM Corporation 2009

Figure 11-2. What have you learned?

AS075.0

### Notes:

As we said at the beginning of this class, this class is the second in a series of three courses that, when followed in sequence, provide the skills necessary to be proficient RPG IV programmer.

You have learned a lot about using the RPG IV language to write applications. To maximize the new skills that you now have, you must apply this knowledge *on the job*.

## To do list

IBM i

- Continue to get lots of hands-on experience
- Participate in RPG and i forums
- Join a local users group
- Attend i Technical Conference

© Copyright IBM Corporation 2009

Figure 11-3. To do list t

AS075.0

### **Notes:**

The most important thing you can do is to write more programs. Use the techniques and tools described in this class to write better functioning and more modular programs.

Check in your area for a local users group and investigate joining it.

IBM i5(iSeries) Technical Conferences cover a wide range of technical topics. Go to:

<http://www-3.ibm.com/services/learning/conf/>

# Topics for the advanced class

IBM i

1. Welcome and administration
2. Basic API programming
3. RPG IV features
4. Leveraging DB2 UDB database features
5. Advanced ILE topics
6. ILE error handling and condition handlers
7. ILE CEE API programming
8. Basic Web enablement
9. What's next?

© Copyright IBM Corporation 2009

Figure 11-4. Topics for the advanced class

AS075.0

## Notes:

You should plan to attend the last RPG IV course in this series of three classes next. Some on the job experience is strongly recommended prior to attending this class.

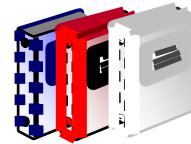
After completing this course, you should be able to:

- Use address pointers and user spaces in RPG IV programs
- Write database triggers in RPG IV
- Develop ILE modular objects and package them in service programs
- Explain the purpose of ILE Activation Groups
- Explain the behavior of Error Handling and Percolation in ILE
- Code an ILE Error Handling program
- Call i5(iSeries) APIs from RPG IV programs
- Describe how to use CGI in an RPG IV application

## Useful references and Web sites

IBM i

- RPG IV reference manuals
- *Who Knew You Could Do That with RPG IV?* (ITSO Redbook)



<http://www.iseriesnetwork.com>

<http://www.midrange.com>

<http://www.rpg-xml.com>

<http://lists.midrange.com/cgi-bin/listinfo/code400-l>

<http://www.ibm.com/certify>

<http://www.rpgiv.com>



© Copyright IBM Corporation 2009

Figure 11-5. Useful references and Web sites

AS075.0

### Notes:

This is a list of some useful documentation as well as some Web sites that you should explore. Of particular value are the forum lists, sponsored by [www.midrange.com](http://www.midrange.com).

# Unit summary

IBM i

Having completed this unit, you should be able to:

- List the prerequisite skills needed prior to attending the Advanced RPG IV classes
- List the topics of the next class

© Copyright IBM Corporation 2009

Figure 11-6. Unit summary

AS075.0

## Notes:





**IBM**  
®