



*RPG IV*  
*Programming Advanced*  
*Workshop for IBM i*  
(Course code AS10)

**Student Notebook**

ERC 6.0

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®	Current®	DB™
DB2®	Integrated Language Environment®	iSeries®
i5/OS™	i5/OS®	Language Environment®
MQSeries®	OS/400®	Power Systems™
Power Systems Software™	Power®	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.

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

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Other product and service names might be trademarks of IBM or other companies.

## **September 2011 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>
Welcome .....	1-2
Administration .....	1-3
Course objectives .....	1-4
Prerequisites .....	1-5
Agenda .....	1-6
Introductions .....	1-7
<b>Unit 2. Basic API programming.</b> .....	<b>2-1</b>
Unit objectives .....	2-2
2.1. Using APIs in RPG IV programs .....	2-3
What are APIs? .....	2-4
Why use APIs? .....	2-5
Categories of APIs .....	2-7
Example: Using QUSCMDLN .....	2-8
Communicating with parameters .....	2-9
API parameters: Required, optional, and omissible .....	2-10
API parameter data type considerations .....	2-11
Example: API parameter definition documentation .....	2-12
Example: API parameter definition: Prototype .....	2-13
Example: Using QCMDEXEC .....	2-15
Asynchronous communications between jobs .....	2-16
Creating data queues .....	2-17
Data queue prototypes .....	2-19
Using QSNTDAQ/QRCVDTAQ .....	2-20
Example: Using QSNDTAQ/QRCVDTAQ (1 of 3) .....	2-21
Example: Using QSNDTAQ/QRCVDTAQ (2 of 3) .....	2-23
Example: Using QSNDTAQ/QRCVDTAQ (3 of 3) .....	2-24
Example: Using QMHSNDPM .....	2-26
Machine exercise: Using system APIs (1 of 2) .....	2-28
Retrieve APIs .....	2-29
Format parameter of retrieve and list APIs .....	2-30
QSYSINC library .....	2-31
Retrieve API example: USROBJD API parameters .....	2-33
Retrieve object description (QUSROBJD) .....	2-34
Example: QUSROBJD .....	2-35
Common problems and error handling .....	2-36
Error handling DS: QUSEC (1 of 2) .....	2-37

Error handling DS: QUSEC (2 of 2) .....	2-38
Other APIs .....	2-40
Machine exercise: Using a system APIs (2 of 2) .....	2-41
Checkpoint (1 of 2) .....	2-42
Checkpoint (2 of 2) .....	2-43
Unit summary .....	2-44
<b>Unit 3. RPG IV features .....</b>	<b>3-1</b>
Unit objectives .....	3-2
<b>3.1. Compiler directives .....</b>	<b>3-3</b>
Compiler directives to control appearance of compile listing .....	3-4
Compiler directive to insert code from another source member .....	3-6
Compiler directives to control which source statements are compiled .....	3-8
/DEFINE and /UNDEFINE .....	3-11
Defining condition names as compilation parameter .....	3-13
Rules for testing conditions .....	3-14
Predefined conditions .....	3-15
Example: Conditional directives (1 of 2) .....	3-16
Example: Conditional directives (2 of 2) .....	3-17
Uses of conditional directives .....	3-18
Machine exercise: Using conditional compiler directives .....	3-19
<b>3.2. Based variables and pointers .....</b>	<b>3-21</b>
Introducing based variables .....	3-22
Manipulating variables using EVAL and BIFs .....	3-23
Manipulating variables Using a data structure .....	3-24
Based variables .....	3-25
Basing pointer data type .....	3-26
Based variable alternative (1 of 2) .....	3-28
Based variable alternative (2 of 2) .....	3-30
<b>3.3. Dynamic storage allocation and pointers .....</b>	<b>3-31</b>
Pointing to heap storage and user spaces .....	3-32
Heap request API example: CEEGTST API parameters .....	3-34
Using the heap: System APIs .....	3-35
Using the heap: RPG IV operations .....	3-36
User space highlights .....	3-39
User space APIs .....	3-40
Creating a user space: QUSCRTUS .....	3-41
Accessing a user space: QUSPTRUS .....	3-42
Contents of My_Space .....	3-43
Using based variables: Big arrays .....	3-44
Problems with pointers .....	3-45
Unresolved pointer .....	3-46
Falling off the edge .....	3-47
Recommendations .....	3-48
<b>3.4. User spaces and list APIs .....</b>	<b>3-49</b>
List APIs .....	3-50
Using list APIs .....	3-51
Output of list API to user space .....	3-52

Generic header format 0100 .....	3-55
Step 1: Create user space .....	3-56
Example: List objects (QUSLOBJ) API parameters .....	3-57
Step 2: Code program to fill user space .....	3-58
Decimal offset .....	3-59
Format OBJL0100 for QUSLOBJ output .....	3-60
Parameters for retrieve user space API (QUSRTVUS) .....	3-61
RPG IV program: Retrieve information from user space (1 of 3) .....	3-62
RPG IV program: Retrieve information from user space (2 of 3) .....	3-63
RPG IV program: Retrieve information from user space (3 of 3) .....	3-64
Retrieve pointer to user space (QUSPTRUS) API .....	3-65
RPG IV program: Retrieve information from a user space using a pointer .....	3-66
Machine exercise: Using list APIs .....	3-68
Checkpoint (1 of 2) .....	3-69
Checkpoint (2 of 2) .....	3-70
Unit summary .....	3-71
<b>Unit 4. ILE CEE API programming .....</b>	<b>4-1</b>
Unit objectives .....	4-2
4.1. An introduction to CEE ILE APIs .....	4-3
What do the ILE CEE APIs provide? .....	4-4
ILE CEE data types .....	4-7
Coding to call a bindable API .....	4-8
CEEGPID example: Parameters .....	4-9
CEEGPID: Retrieve current platform and release level .....	4-10
Operational descriptors .....	4-12
CEEDAYS: Get Lilian date parameters .....	4-13
ILE feedback code .....	4-15
CEEDAYS example (1 of 2) .....	4-18
CEEDAYS example (2 of 2) .....	4-19
CEELOCT: Get current local time parameters .....	4-20
CEELOCT example (1 of 2) .....	4-22
CEELOCT example (2 of 2) .....	4-23
Math functions .....	4-24
CEE4SxFAC: N factorial .....	4-25
CEE4SIFAC example .....	4-26
CEESxMOD: Modular arithmetic .....	4-27
CEESIMOD example .....	4-28
CEETSTA: Test arguments .....	4-29
CEETSTA example .....	4-30
*PGM and *SRVPGM information .....	4-31
Machine exercise: Using bindable CEE APIs .....	4-33
Checkpoint .....	4-34
Unit summary .....	4-35
<b>Unit 5. Leveraging DB2 UDB database features .....</b>	<b>5-1</b>
Unit objectives .....	5-2
5.1. Commitment control .....	5-3

DB journaling: Protection against loss .....	5-4
Implementing journaling .....	5-6
Journal codes .....	5-7
Functions of commitment control .....	5-8
Transaction definition .....	5-10
Establishing a commitment control definition .....	5-11
Specify files for commitment control open .....	5-12
Commitment control considerations .....	5-13
Journal receiver entries .....	5-14
COMMIT: All or nothing .....	5-15
Commit/rollback .....	5-16
Syntax of COMMIT/ROLBK operations .....	5-17
Calculations for commitment control .....	5-19
End of program .....	5-20
Automatic rollback occurs .....	5-21
Notify object .....	5-22
How can a notify object be used? .....	5-24
When is a notify object updated? .....	5-26
Restart logic .....	5-27
Commitment control with notify object example (1 of 6) .....	5-28
Commitment control with notify object example (2 of 6) .....	5-29
Commitment control with notify object example (3 of 6) .....	5-30
Commitment control with notify object example (4 of 6) .....	5-31
Commitment control with notify object example (5 of 6) .....	5-32
Commitment control with notify object example (6 of 6) .....	5-33
Using Notify Object Recovery .....	5-34
<b>5.2. Database triggers .....</b>	<b>5-35</b>
What is a trigger? .....	5-36
Triggers: Introduction .....	5-38
Triggers: An example .....	5-40
Trigger components .....	5-41
Adding a trigger to a file .....	5-42
Other trigger commands .....	5-45
The base file .....	5-46
Trigger time: BEFORE and AFTER .....	5-48
Trigger time example: Triggers before and after a change operation .....	5-49
The trigger event .....	5-51
The trigger program .....	5-52
Trigger buffer: Fixed portion of parameter .....	5-53
Trigger buffer: Variable portion of parameter .....	5-55
Coding example: Trigger parameter list .....	5-56
Coding example: Trigger program: Hard-coded buffer .....	5-58
Coding sample: Trigger program: Better method .....	5-60
Managing errors in triggers .....	5-62
Data integrity and trigger failures .....	5-64
Updating the record .....	5-66
Order of execution .....	5-67
Trigger can perform update cascade .....	5-69

Machine exercise: Database triggers .....	5-70
5.3. Embedded SQL: An overview .....	5-71
Embedded SQL: Some questions .....	5-72
Coding choices .....	5-73
Why use embedded SQL? .....	5-75
What Is embedded SQL? .....	5-77
Inquiry program: Native I/O .....	5-79
Inquiry program: SQL I/O .....	5-82
Program generation .....	5-83
Embedded SQL: Under the covers (1 of 2) .....	5-85
Embedded SQL: Under the covers (2 of 2) .....	5-86
Taking advantage of embedded SQL .....	5-87
Checkpoint (1 of 2) .....	5-89
Checkpoint (2 of 2) .....	5-90
Unit summary .....	5-91
<b>Unit 6. Advanced ILE topics .....</b>	<b>6-1</b>
Unit objectives .....	6-2
6.1. Packaging modules .....	6-3
Review .....	6-4
Calling program calls called program .....	6-6
CONST and OPTIONS keywords for parameter definition .....	6-7
Example: Basic subprocedure .....	6-9
Modular programming .....	6-11
Program with many modules .....	6-12
Module can contain more than one procedure .....	6-13
Subprocedures with NOMAIN procedure .....	6-15
Today's RPG IV program .....	6-16
EXPORT keyword on P-specification .....	6-17
Subprocedure packaged in same module as main procedure: Example .....	6-18
Call NbrDays from another procedure .....	6-19
Add EXPORT to NbrDays .....	6-20
Packaging subprocedures in a NOMAIN procedure .....	6-21
Export considerations .....	6-22
Subprocedures in a NOMAIN procedure .....	6-23
Steps required to make subprocedures available .....	6-25
Bind subprocedures in service program .....	6-26
Summary of application .....	6-27
Machine exercise: Enhancing NOMAIN service program .....	6-28
6.2. Binder language .....	6-29
What is binder language? .....	6-30
Using binder language .....	6-32
RTVBNDSRC .....	6-34
Output of RTVBNDSRC .....	6-35
Signature of service program .....	6-37
Display service program's signature .....	6-39
The power of binder language and signatures (1 of 7) .....	6-40
The power of binder language and signatures (2 of 7) .....	6-41

The power of binder language and signatures (3 of 7) . . . . .	6-42
The power of binder language and signatures (4 of 7) . . . . .	6-43
The power of binder language and signatures (5 of 7) . . . . .	6-44
The power of binder language and signatures (6 of 7) . . . . .	6-45
The power of binder language and signatures (7 of 7) . . . . .	6-46
Managing multiple signatures . . . . .	6-47
Binder language and scope of data export . . . . .	6-48
<b>6.3. Binding directories . . . . .</b>	<b>6-51</b>
What is a binding directory? . . . . .	6-52
Binding directory . . . . .	6-53
Using a binding directory . . . . .	6-55
WRKBNNDDIR . . . . .	6-56
WRKBNNDDIRE . . . . .	6-57
Binding directory entry added . . . . .	6-58
Using a binding directory . . . . .	6-59
Machine exercise: Using binding directories and binder language . . . . .	6-61
<b>6.4. Activation groups . . . . .</b>	<b>6-63</b>
Activation groups . . . . .	6-64
What are activation groups? . . . . .	6-65
Using activation groups . . . . .	6-66
What is activation? . . . . .	6-67
Application isolation . . . . .	6-68
Sample activation . . . . .	6-69
Creating activation groups . . . . .	6-72
Deleting an activation group . . . . .	6-75
Activation group scoping . . . . .	6-77
Types of scoping . . . . .	6-79
File open scoping . . . . .	6-81
File override scoping . . . . .	6-82
Commitment control scoping . . . . .	6-84
<b>6.5. Other ILE matters . . . . .</b>	<b>6-85</b>
Recursive calls of subprocedures . . . . .	6-86
Recursion: An example . . . . .	6-87
Encapsulating SQL based I/O In service programs . . . . .	6-88
Development process . . . . .	6-89
Using embedded SQL: Existing program example . . . . .	6-90
Using embedded SQL: I/O module . . . . .	6-91
Using embedded SQL: Main procedure . . . . .	6-92
Using embedded SQL: Commands used to create application . . . . .	6-93
Checkpoint (1 of 2) . . . . .	6-94
Checkpoint (2 of 2) . . . . .	6-95
Unit summary . . . . .	6-96
<b>Unit 7. ILE error handling and condition handlers . . . . .</b>	<b>7-1</b>
Unit objectives . . . . .	7-2
<b>7.1. An overview of ILE error handling. . . . .</b>	<b>7-3</b>
Error handling . . . . .	7-4
Types of error messages . . . . .	7-6

Job message queues .....	7-7
Exception handlers in RPG IV .....	7-8
OPM traditional exception handling .....	7-9
ILE control boundaries .....	7-11
Where are the control boundaries? .....	7-12
What happens at a control boundary? .....	7-13
Percolation .....	7-14
Unhandled ILE exception .....	7-15
ILE exception handling example .....	7-17
OPM versus ILE error handling .....	7-19
ILE exception handling .....	7-20
ILE error handling .....	7-22
Conversion considerations .....	7-23
Using ILE error handlers .....	7-24
ILE condition handler .....	7-26
ILE cancel handlers .....	7-27
<b>7.2. ILE condition handlers</b> .....	<b>7-29</b>
An overview of using ILE condition handlers .....	7-30
ILE condition handling: Registering the handler .....	7-31
ILE condition handling: Handling an error .....	7-33
ILE condition handling: Resume point .....	7-34
Registration of condition handler .....	7-35
Procedure interface: Condition handler .....	7-37
Condition token .....	7-39
Error message severity .....	7-41
Actions passed to ILE Condition Manager .....	7-43
Condition handler: CONDHDLR (1 of 2) .....	7-45
Condition handler: CONDHDLR (2 of 2) .....	7-47
Main procedure: CauseErr (1 of 2) .....	7-48
Main procedure: CauseErr (2 of 2) .....	7-49
Creating PGM with condition handler .....	7-50
ILE condition handling: The process in review .....	7-51
Condition handler: Magic? .....	7-52
Machine exercise: Enhancing the condition handler .....	7-54
Checkpoint .....	7-55
Unit summary .....	7-56
<b>Unit 8. Other RPG IV compiler features</b> .....	<b>8-1</b>
Unit objectives .....	8-2
V6R1 enhancements .....	8-3
Main subprocedure .....	8-5
Files in subprocedures .....	8-6
V7R1 enhancements .....	8-8
Unit summary .....	8-12
<b>Appendix A. Checkpoint solutions</b> .....	<b>A-1</b>
<b>Bibliography</b> .....	<b>X-1</b>



# 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®	Current®	DB™
DB2®	Integrated Language Environment®	iSeries®
i5/OS™	i5/OS®	Language Environment®
MQSeries®	OS/400®	Power Systems™
Power Systems Software™	Power®	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.

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

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Other product and service names might be trademarks of IBM or other companies.



# Course description

## RPG IV Programming Advanced Workshop for IBM i

**Duration:** 4 days

### Purpose

This course teaches additional skills and techniques to programmers who can already write comprehensive RPG IV programs.

This class offers a comprehensive discussion of some of the advanced features and functions of RPG IV. This class is designed to enable an experienced RPG IV programmer to develop and maintain RPG IV programs of an advanced level using the latest features and techniques available in the IBM i RPG IV compiler.

### Audience

This course is the third in a series of three classes designed for programmers who want to learn to code using the IBM i ILE RPG IV language. A previous programming experience using RPG IV is mandatory before enrolling in this course. The student should have attended AS06/AS060 and AS07/AS070. Students who have attended S6126 also meet the prerequisites for this course.

This class is *not* designed for RPG III programmers who want to learn RPG IV. RPG III programmers should attend the *Moving from RPG/400 to RPG IV for System i* class (OE85/OE850) before taking this class. RPG III programmers should review the agenda carefully before they make a decision to attend this class.

**Note:** The term RPG/400 refers to both System/38 RPG as well as AS/400 RPG/400 (aka RPG III).

Previous techniques and the maintenance of programs written using legacy techniques are not covered in the classroom.

### Prerequisites

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

- Use a Windows-based PC
- Run PC applications using menus, icons, tool bars, and so forth

The following skills are taught in OL49/OL490:

- Use IBM i navigation tools, including:

- Use and prompt command line (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 print queues
- Use CL commands such as WRKJOB and DSPMSG
- Use Rational Developer for Power Systems (RDP) or Program Development Manager (PDM) and Source Entry Utility (SEU) to create and maintain RPG IV source members
- Create and maintain physical and logical files

The following skills are taught in AS06/AS060:

- Write RPG IV programs to produce reports
- Write simple RPG IV inquiry programs that interact with displays
- Compile RPG IV programs
- Review compilation listing and find and correct compilation errors
- Maintain existing applications written in the RPG IV language
- Use the Debugger tool to determine the cause of incorrect results
- Use many popular RPG IV built-in functions

The following skills are taught in AS07/AS070:

- Use OVERLAY and related data description specification (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
- 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

Students must have attended these courses prior to attending this class:

- *Introduction to IBM i for New Users* (OE98/OE980)
- *System i Application Programming Facilities Workshop* (OL49/OL490)
- *RPG IV Programming Fundamentals for IBM i Workshop* (AS06/AS060)
- *RPG IV Programming Intermediate for IBM i Workshop* (AS07/AS070)

or have equivalent experience

Attendance at *IBM i Application Development Using WDSC* (OW870) is strongly recommended prior to attending this class.

## Objectives

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 application program interfaces (APIs) from RPG IV program

## Contents

1. Welcome and administration
2. Basic API programming
3. RPG IV features
4. ILE CEE API programming
5. Leveraging DB2 UDB database features
6. Advanced ILE topics
7. ILE error handling and condition handlers
8. Other RPG IV compiler features

## Curriculum relationship

This class is the third of three classes designed to develop RPG IV programmers. It is part of the IBM i programming curriculum to support the RPG IV compiler. Below are other courses 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:

- OW870 - *IBM i Application Development using WDSC*
- OD47.OD470 - IBM i DB2 and SQL School
- OL37/OL370 - *Accessing the IBM i Database Using SQL*
- OL38/OL380 - *Developing System i Applications Using SQL*
- OL39/OL390 - *DB2 UDB for System i SQL Advanced Programming*

# Agenda

## Day 1

- Unit 1 - Using Subprocedures
- Unit 2 - Basic API programming
- Unit 3 - RPG IV features

## Day 2

- Unit 4 - ILE CEE API programming
- Unit 5 - Leveraging DB2 UDB database features

## Day 3

- Unit 5 - Leveraging DB2 UDB database features (continued)
- Unit 6 - Advanced ILE topics

## Day 4

- Unit 6 - Advanced ILE topics (continued)
- Unit 7 - ILE error handling and condition handlers
- Unit 8 - Other RPG IV compiler features



# 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.

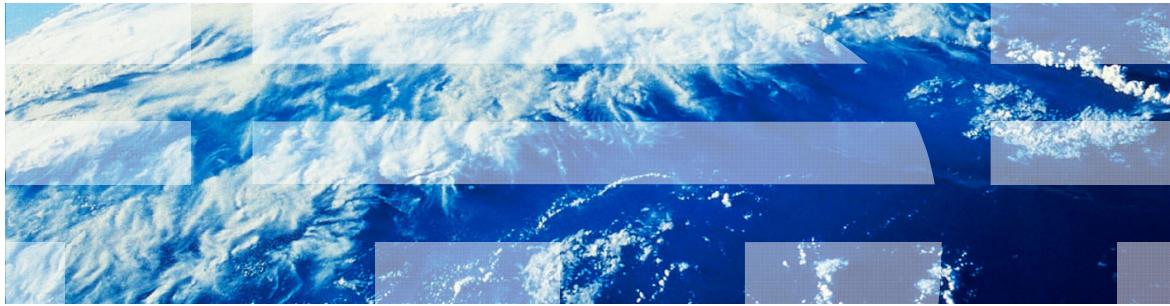
## What you should be able to do

After completing this 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 IBM i APIs from RPG IV programs
- Call APIs from RPG IV programs
- Describe how to use CGI in an RPG IV application



# RPG IV Programming Advanced Workshop for IBM i



***Power<sup>tm</sup> with IBM i***

© Copyright IBM Corporation 2011

Course materials may not be reproduced in whole or in part without the prior written permission of IBM.

6.0

Figure 1-1. Welcome

AS106.0

## **Notes:**

This class is the third in the series of three RPG IV classes. We hope that you have been using the RPG IV skills that you learned in AS07/AS070, *RPG IV Programming Intermediate Workshop for i*.

# Administration

IBM i

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

© Copyright IBM Corporation 2011

---

Figure 1-2. Administration

AS106.0

## **Notes:**

## Course objectives

IBM i

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 IBM i APIs from RPG IV programs

© Copyright IBM Corporation 2011

Figure 1-3. Course objectives

AS106.0

### Notes:

We continue at the point we left off at the end of the AS07/AS070 class.

# Prerequisites

IBM i

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

- Run PC applications using menus, icons, tool bars, and so forth
- Use CL commands
- Use Program Development Manager (PDM), Source Entry Utility (SEU) (or LPEX Editor) to create and maintain RPG IV source members
- Create and maintain physical and logical files
- Write RPG IV programs that:
  - Handle batch and interactive I/O
  - Use arrays and data structures in RPG IV programs
  - Include error and exception handling
  - Use prototyping to call other programs
  - Call subprocedures
- Use IBM i debugging facilities
- Write ILE modules and include those modules in program objects using bind by copy and bind by reference

© Copyright IBM Corporation 2011

Figure 1-4. Prerequisites

AS106.0

## Notes:

You should have attended AS07/AS070 before this class.

# Agenda

IBM i

## Day 1

- Unit 1: Using SubProcedures
- Unit 2: An Introduction to the ILE
- Unit 3: Basic API programming

## Day 2

- Unit 4: RPG IV features
- Unit 5: ILE CEE API programming
- Unit 6: Leveraging DB2 UDB database features

## Day 3

- Unit 6: Leveraging DB2 UDB database features
- Unit 7: Advanced ILE topics (subprocedures and ILE review)

## Day 4

- Unit 8: ILE error handling and condition handlers
- Unit 9: Other RPG IV compiler features

© Copyright IBM Corporation 2011

---

Figure 1-5. Agenda

AS106.0

## Notes:

# Introductions

IBM i

- Your name and city
- Company/Organization name
- Previous programming experience
- Expectations



© Copyright IBM Corporation 2011

Figure 1-6. Introductions

AS106.0

## Notes:

Please introduce yourself. We are very interested in your programming background and the RPG IV programming that you have done since you attended AS07/AS070.



# Unit 1. 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.

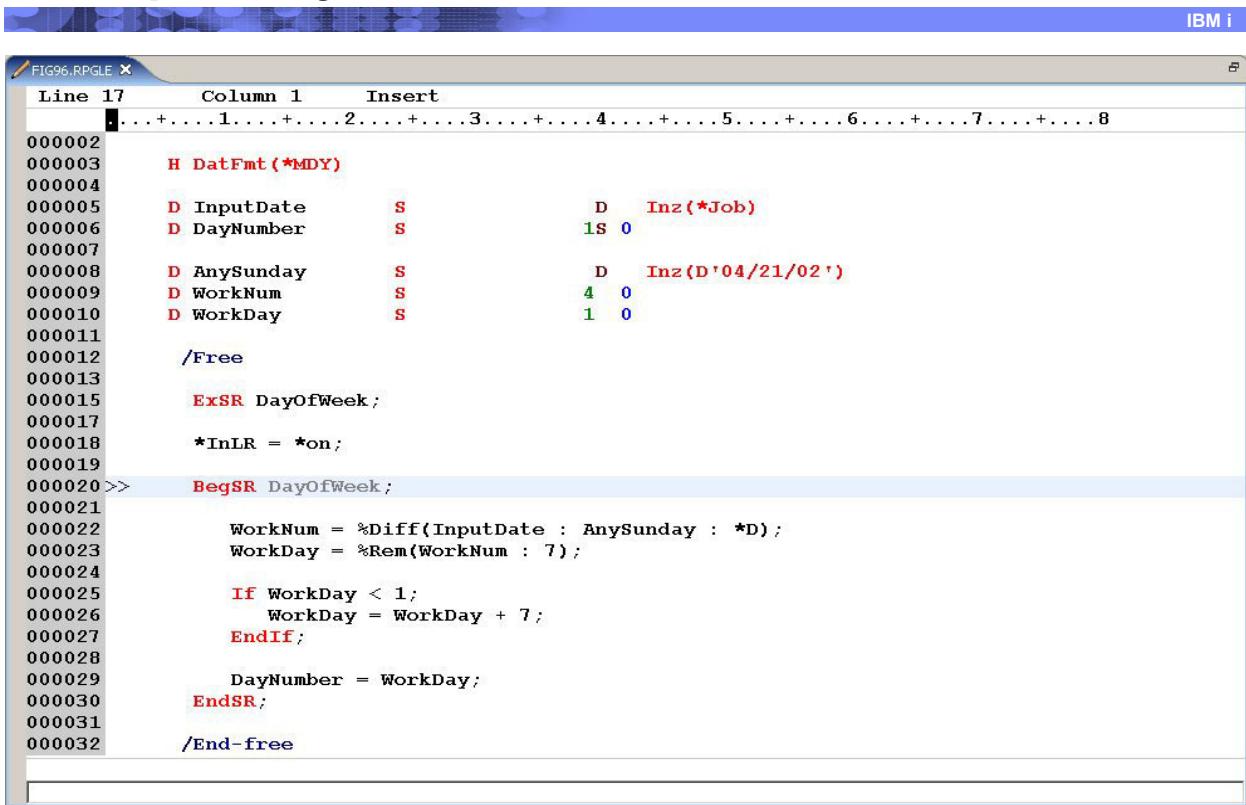
A *subprocedure* is a subprogram that is contained within a procedure or packaged in an ILE Service Program. An RPG IV *subprocedure* is the logical evolution of an RPG subroutine.

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?

```

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
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
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

```

FIG97.RPGLE X
Line 3      Column 1      Replace
1. .... /1....+....2....+....3....+....4....+....5....+....6....+....7....+....8....+....9....+....0
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

```

© Copyright IBM Corporation 2009

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           /Free
00013
00014           1
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	2	D DayOfWeek	PI
00024	2	D InpDate	
00025	2	:	
00040	2	:	
00041	2	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	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	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**).

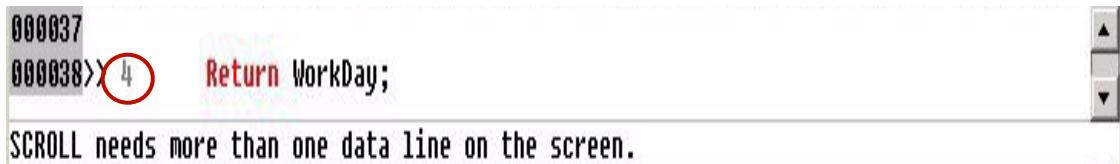
Local data is significant in your implementation of code that is reusable. It is also important to make maintenance tasks less prone to error, because there is more variable independence within the subprocedures and, therefore, less chance that maintenance tasks will inadvertently affect your logic.

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

```

I00100  D DayOfWeek      PR      1S 0
I00200  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
000016
000017  WorkDay = DayOfWeek(InpDate);
000018  Return DayArray(WorkDay);
000019
000020
000021  /End-Free
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

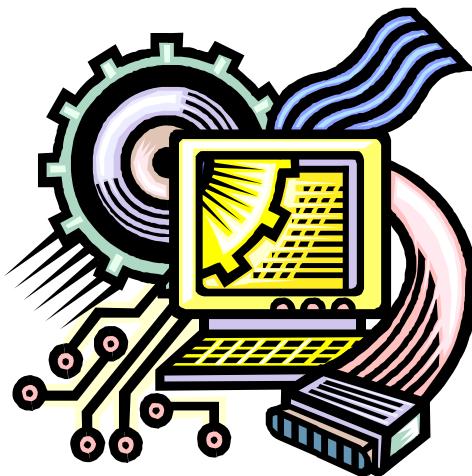
AS075.0

## 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 2. 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

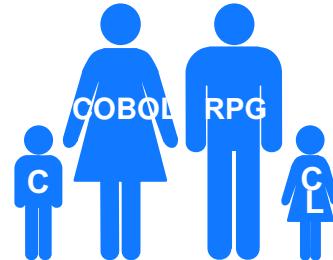
## **Notes:**

## 2.1. 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.





## 2.2. 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.

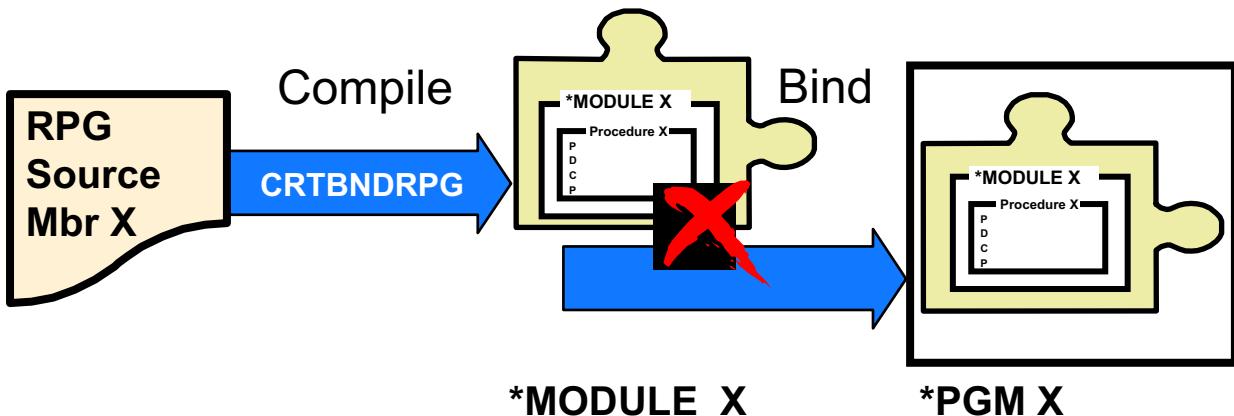
There are other ways to call programs that require less initiation time. Before we discuss these different calls, it is necessary to understand the terms used to describe objects and processes in ILE.

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.

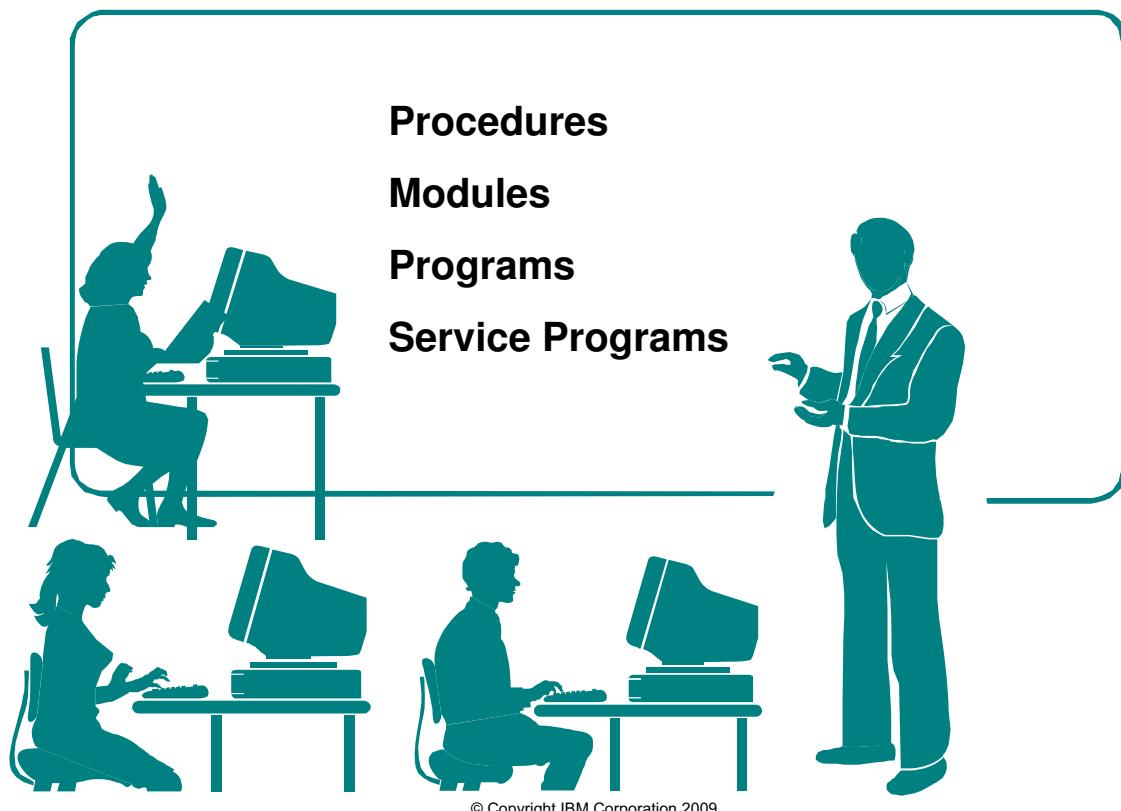
CRTBNDRPG provides a simple, *single-step* approach to program creation that provides very limited use of ILE facilities. There is a one-to-one relationship of:

**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:

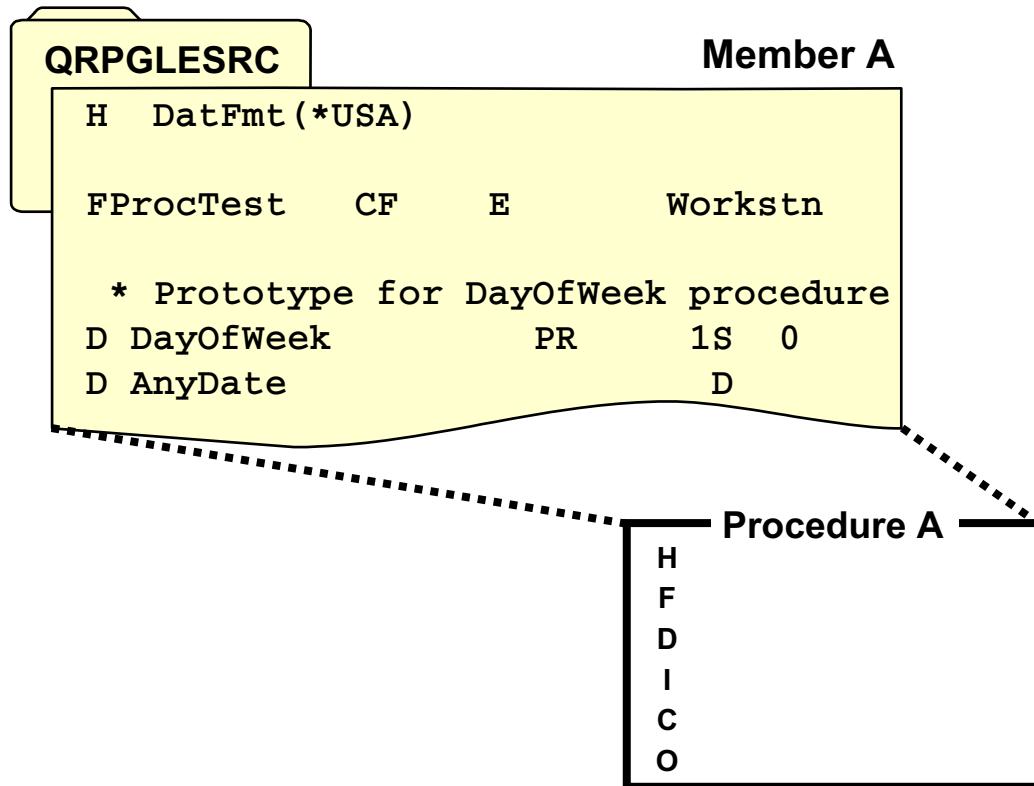
A procedure contains the entry point into the compiled code.

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*.

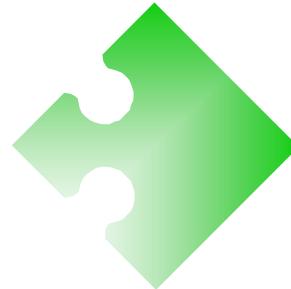
For RPG IV, you would code a source member of type **RPGLE**.

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.

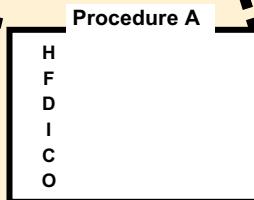
## Module (2 of 2)

IBM i

### Member A

```
H DatFmt(*USA)
FProcTest CF E Workstn
* Prototype for DayOfWeek procedure
D DayOfWeek PR 1S 0
D AnyDate D
```

\*MODULE A



Compile

**CRTRPGMOD**

© Copyright IBM Corporation 2009

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:

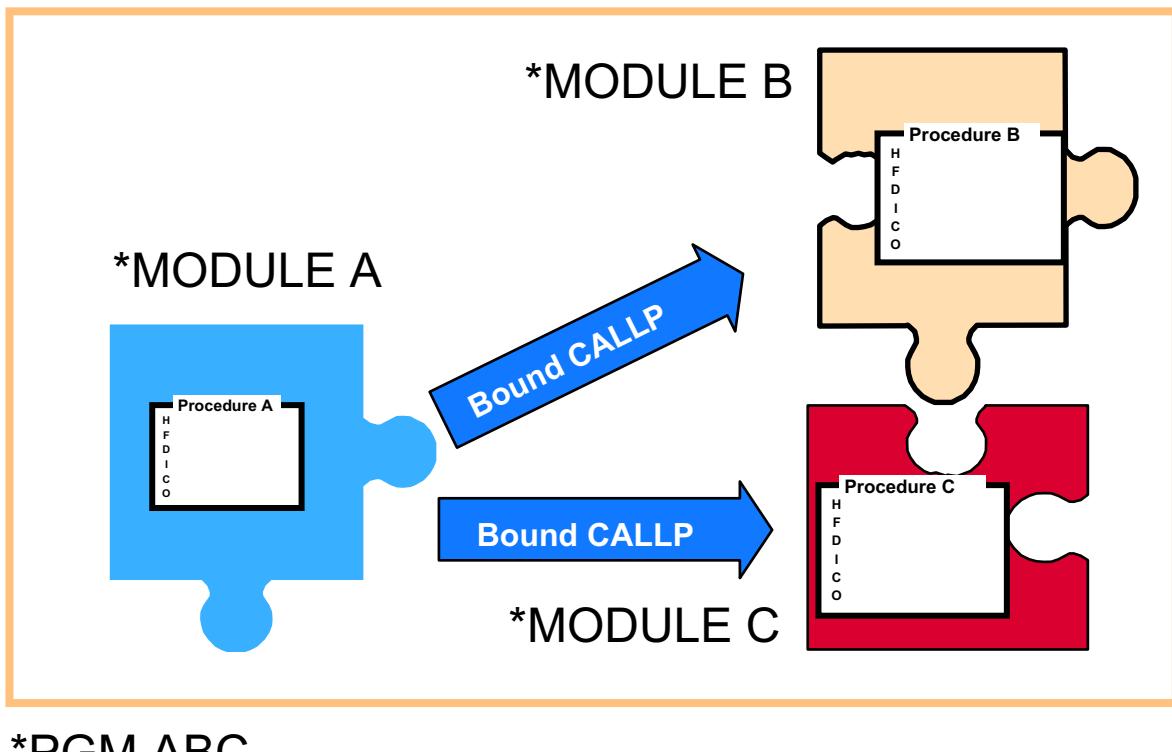
Programs are still called in exactly the same way they are in the OPM environment that you have been using. In some cases, their behavior is different if they are ILE programs.

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.

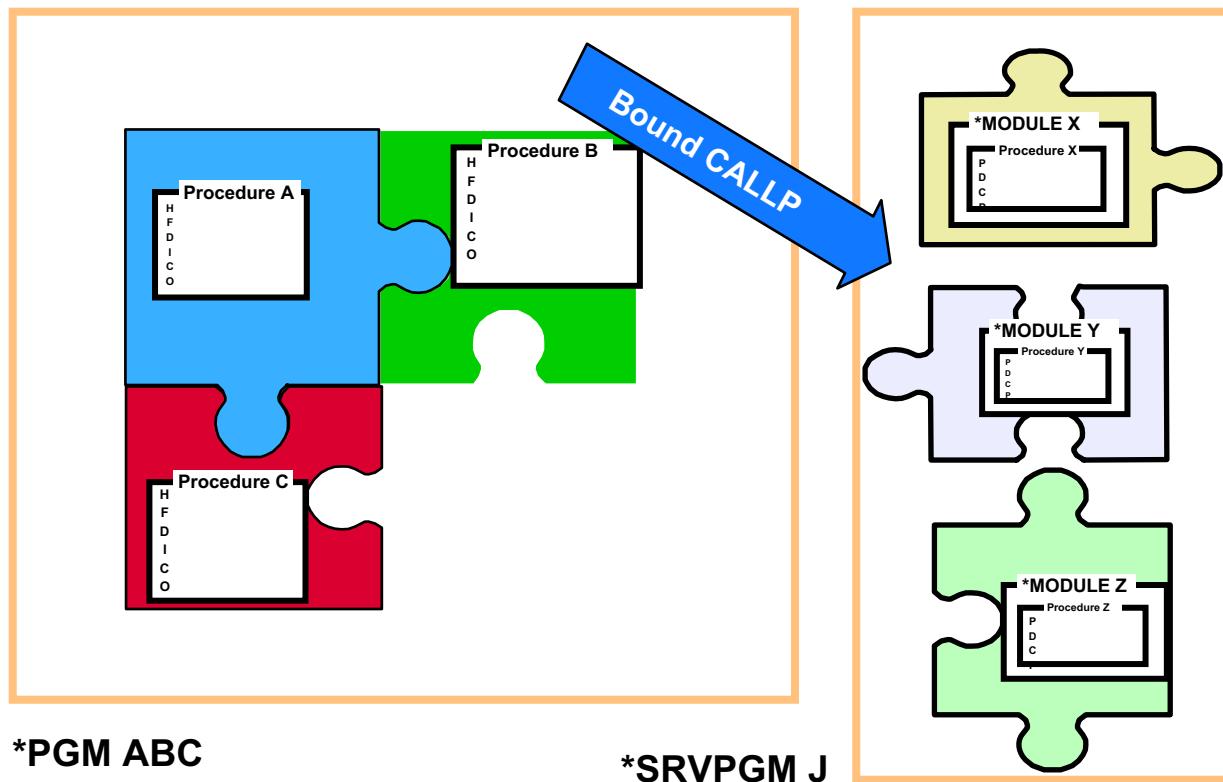
When a program is created that calls a procedure contained in a service program, the program is said to be *bound by reference* to the service program.

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

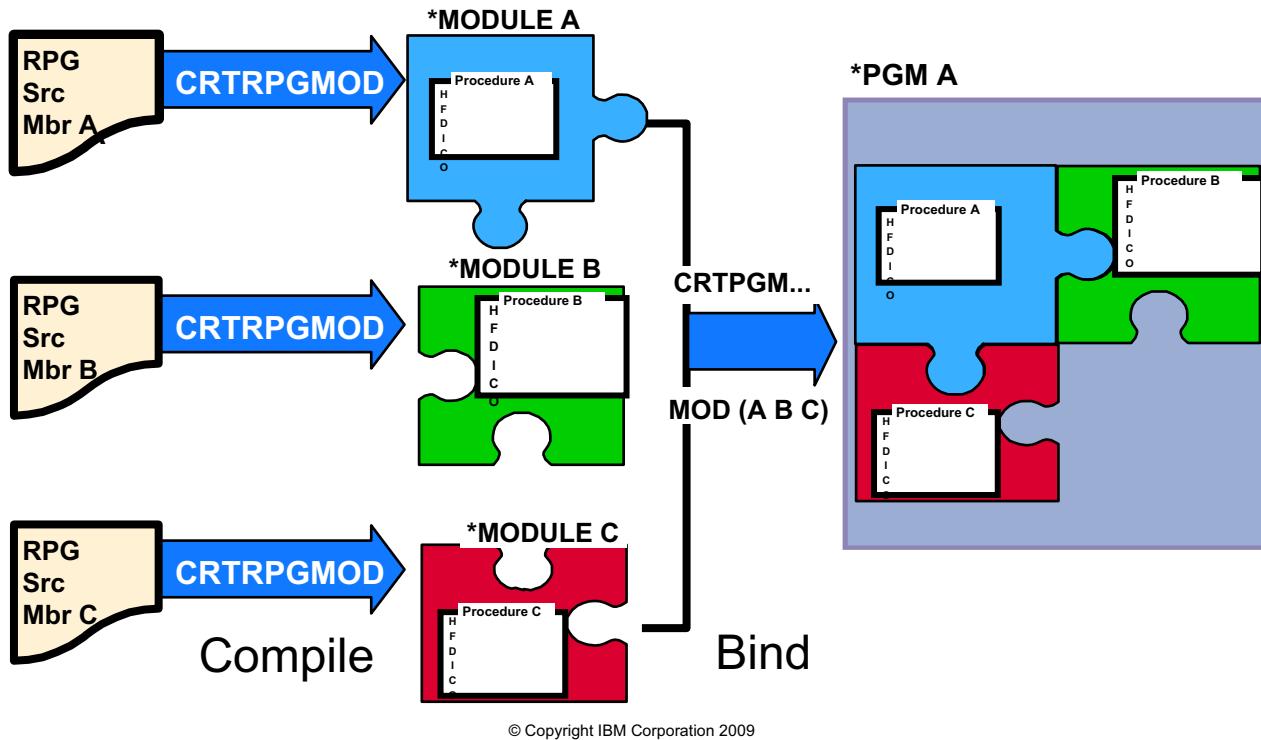


Figure 10-16. Creating ILE RPG IV programs (4 of 4)

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

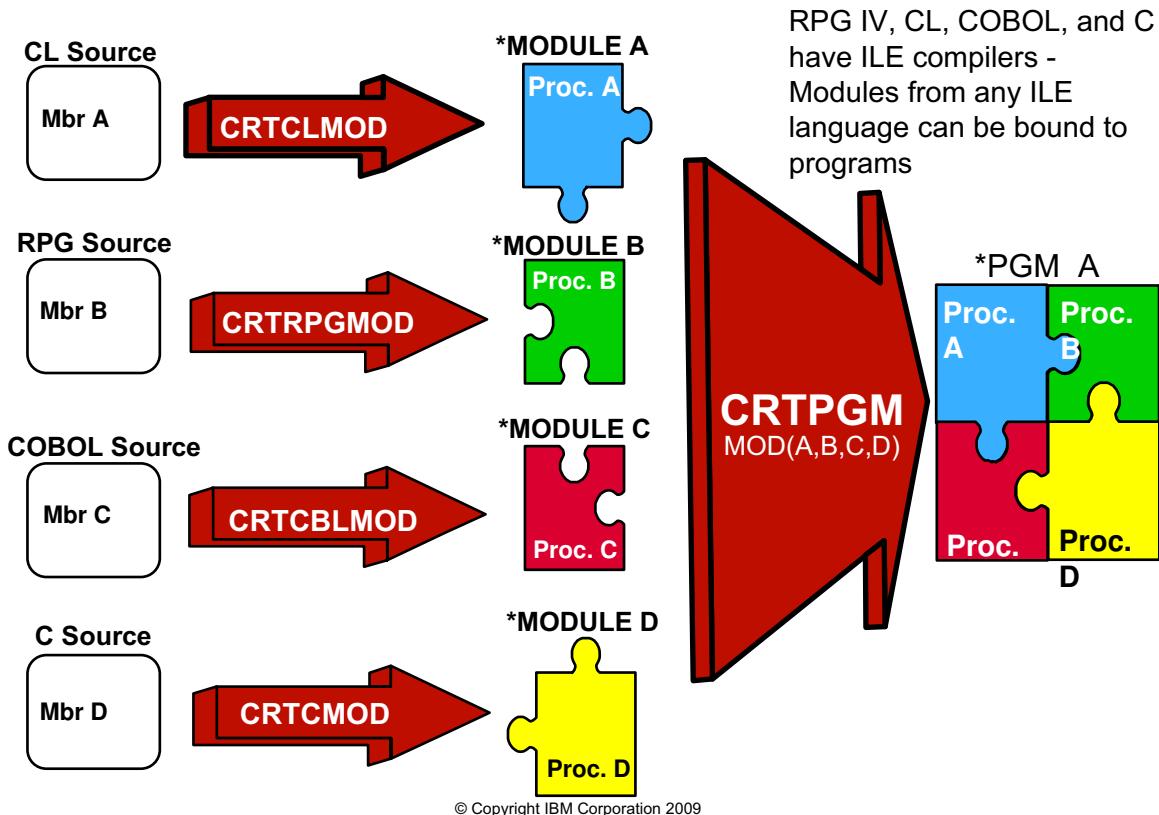


Figure 10-17. Creating ILE programs: Bind by copy

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.

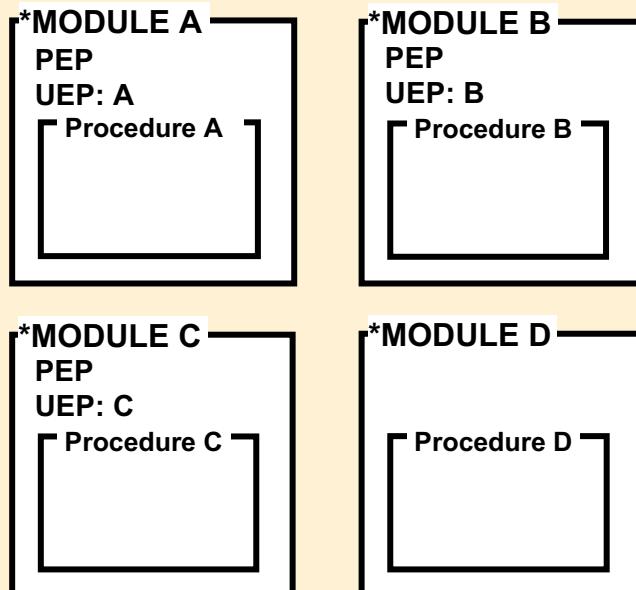
In a \*PGM, many modules have UEPs. You specify which one is the UEP for your \*PGM object in the ENTMOD parameter when you run **CRTPGM**.

# 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

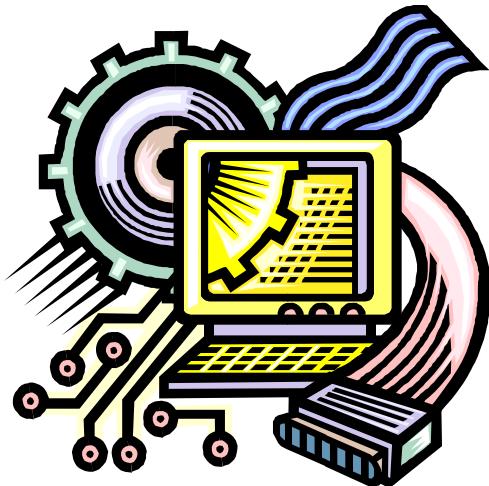
AS075.0

### 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

## 2.3. 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

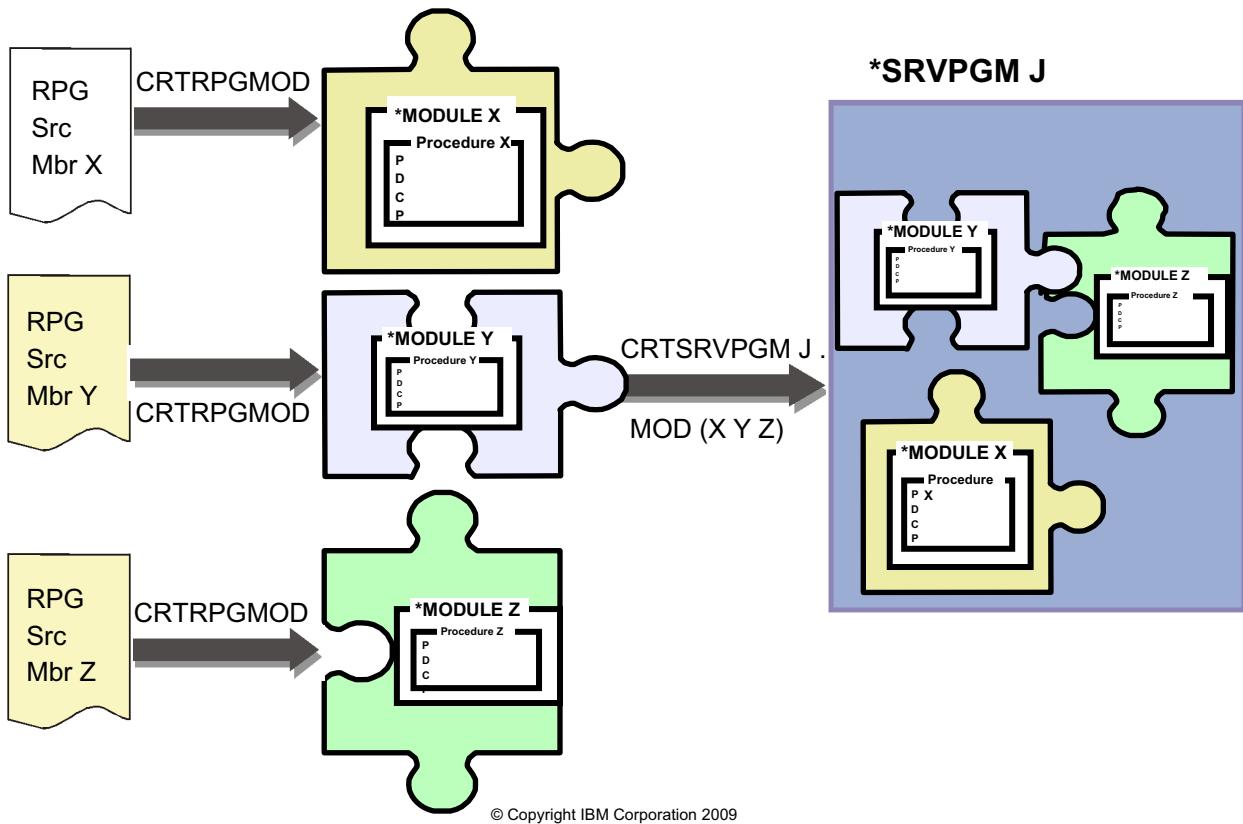


Figure 10-22. Creating a service program

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.

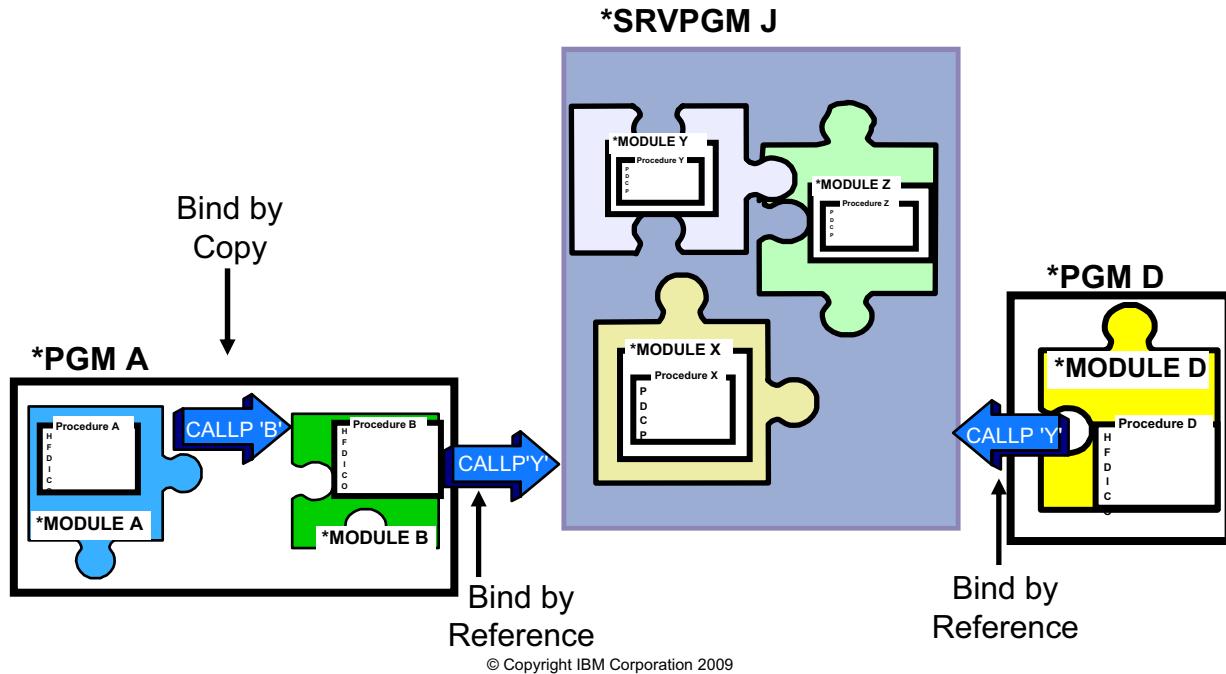


Figure 10-24. Using service programs

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

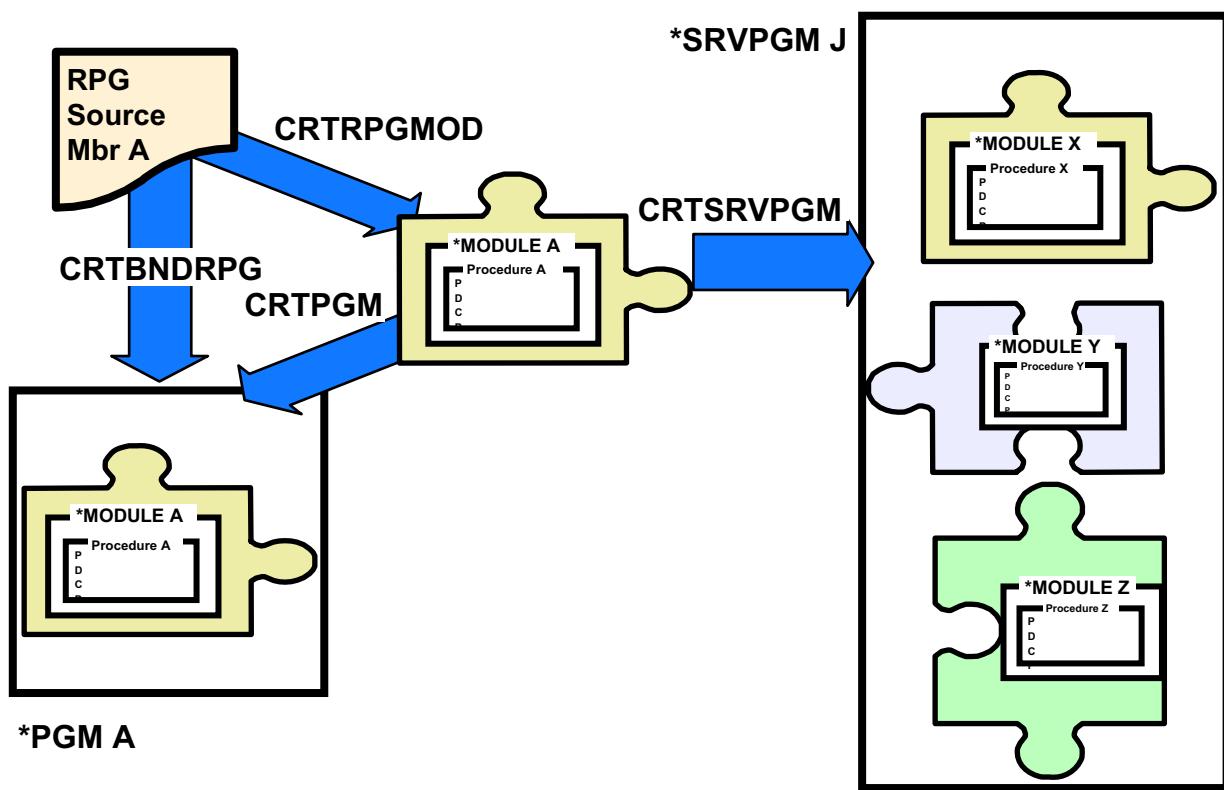
## Notes:

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:

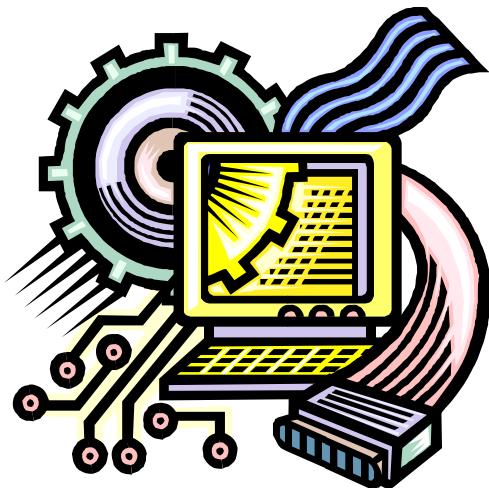
This visual summarizes the different ways that you can create ILE objects by using different commands.

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.”

## 2.4. 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

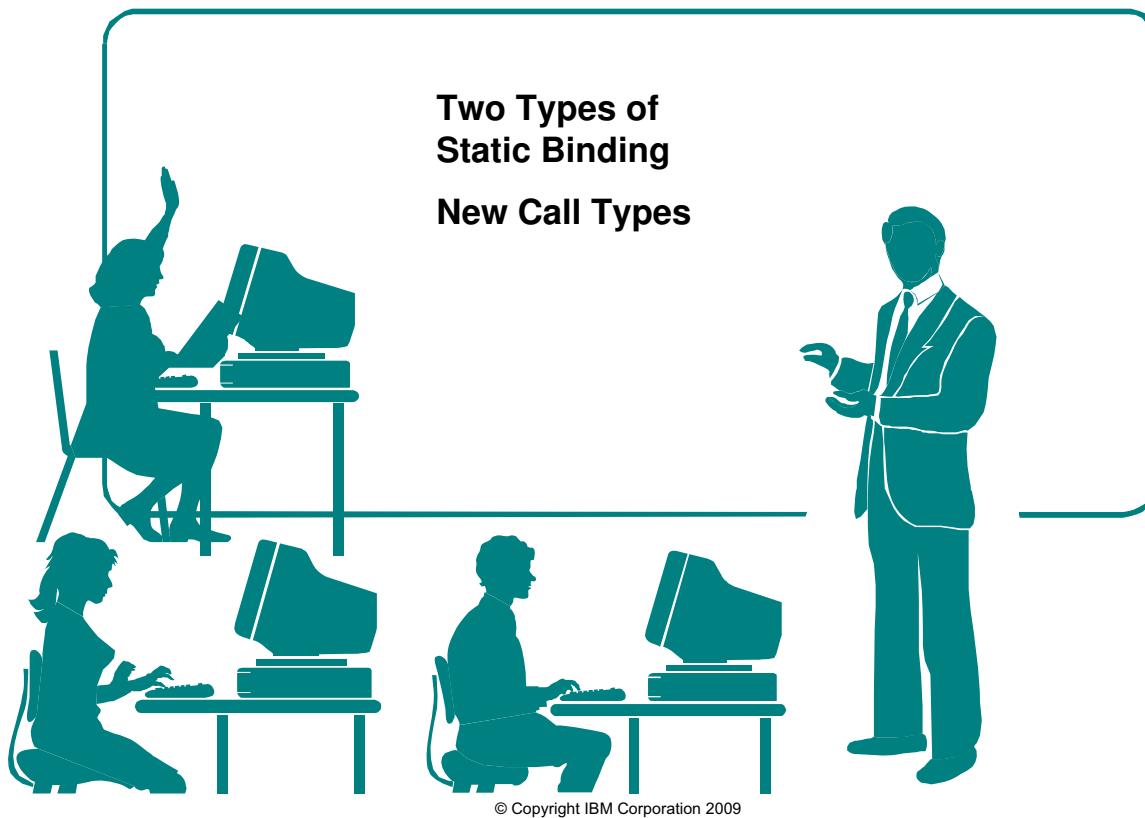


Figure 10-31. Static binding

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).

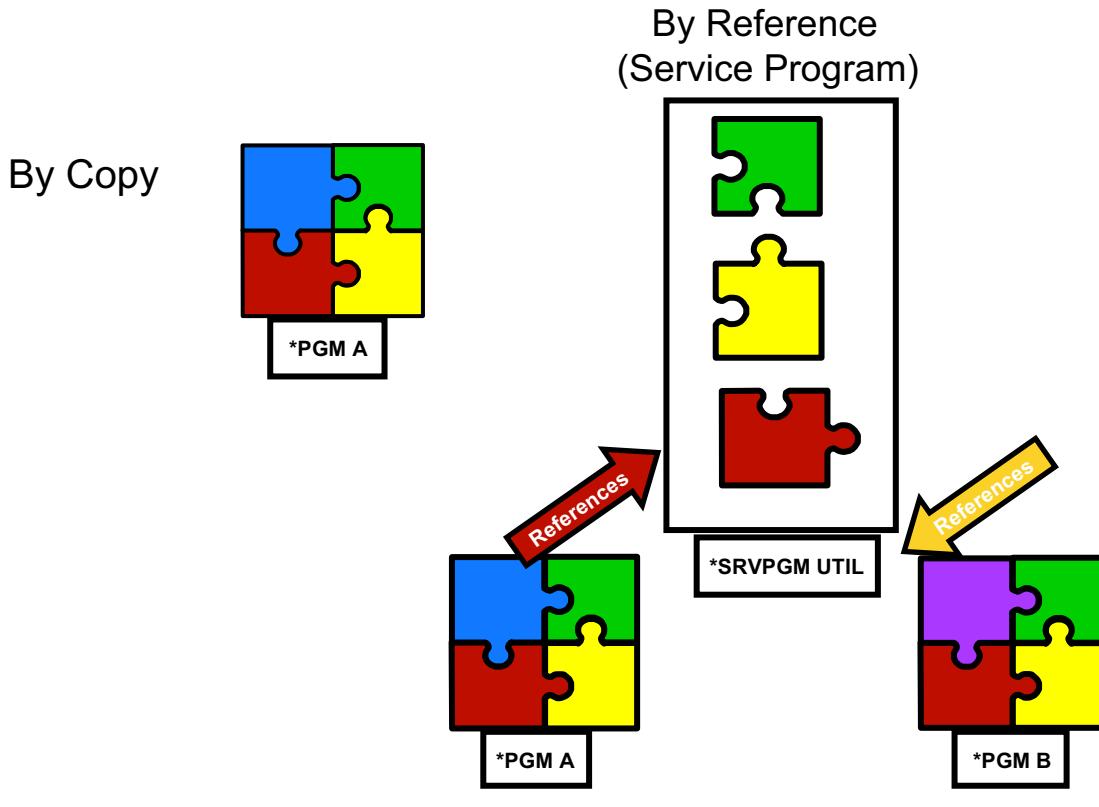
In the world of ILE, we need to be specific in describing the type of bind that is being performed. To simply use the term *bind* is no longer sufficient.

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**

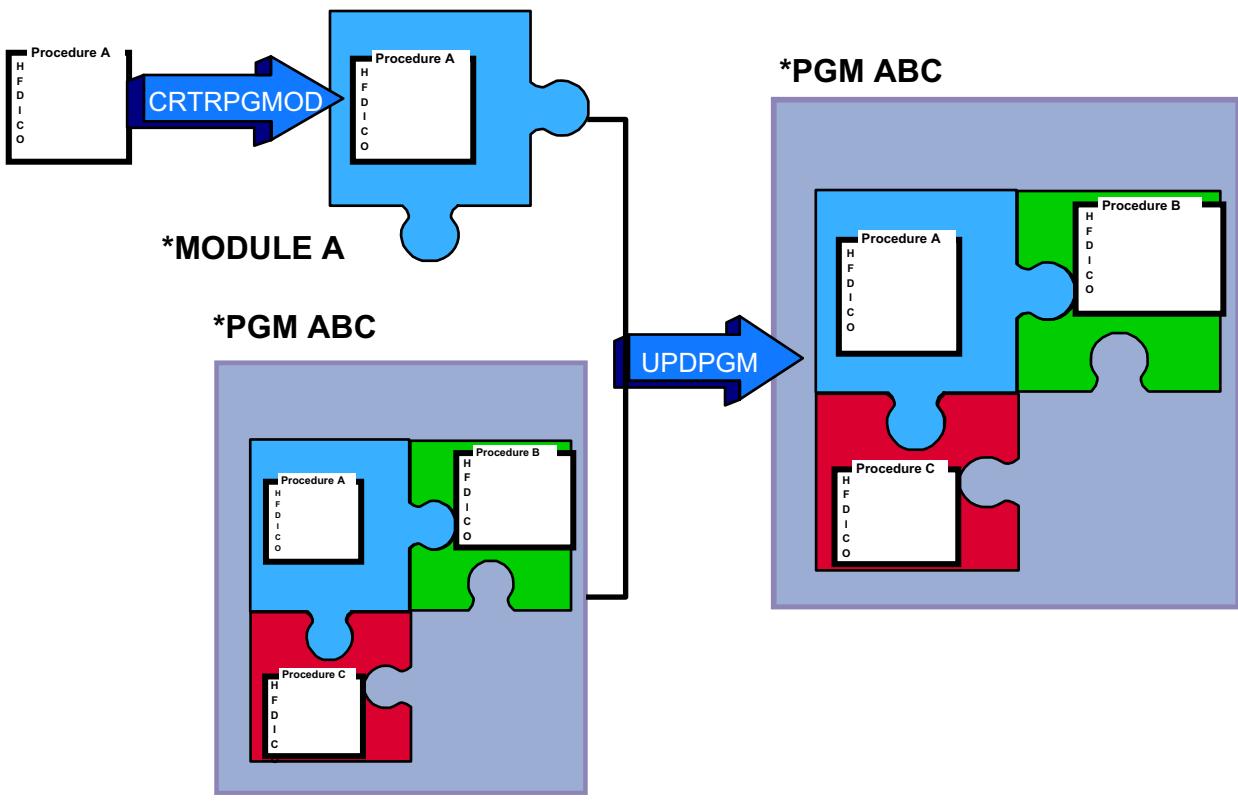
Bind by reference provides the same fast call support as bind by copy. However, there is now a minimum of one extra object involved (one or more Service Programs); so the application's structure becomes somewhat more complex. When a module of code is used by many programs, the Service Program provides 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



© Copyright IBM Corporation 2009

Figure 10-34. Updating a \*PGM

AS075.0

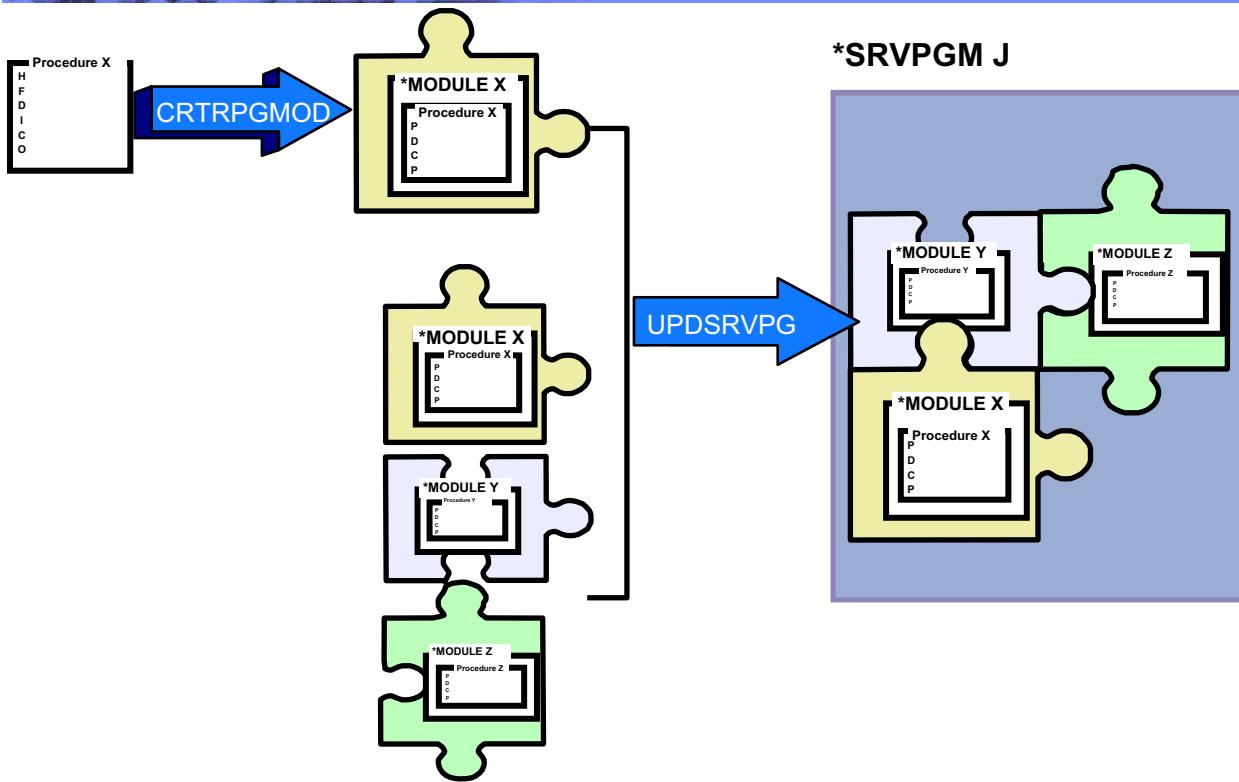
## Notes:

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

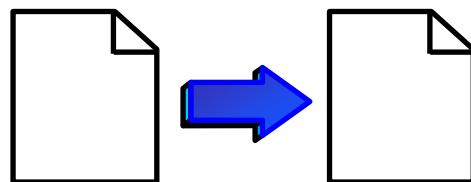
## Notes:

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 3. Basic API programming

## What this unit is about

This unit describes the use of APIs to enhance your RPG IV applications.

## What you should be able to do

After completing this unit, you should be able to:

- Find API documentation in the i Information Center
- Determine parameters and prototype definition necessary to call a system API
- Code RPG IV programs that call system APIs

## How you will check your progress

Accountability:

- Machine exercises
- Checkpoint questions

## Unit objectives

IBM i

After completing this unit, you should be able to:

- Find API documentation in the i Information Center
- Determine parameters and prototype definition necessary to call a system API
- Code RPG IV programs that call system APIs

© Copyright IBM Corporation 2011

Figure 2-1. Unit objectives

AS106.0

### **Notes:**

### **3.1. Using APIs in RPG IV programs**

## What are APIs?

IBM i

- Application programming interface
- System-supplied program that can be called from high-level language program
- Performs tasks you might use CL commands to accomplish
- Examples:
  - [QCMDEXC \(Execute command\)](#)
  - [QSNDATAQ \(Send to a data queue\)](#)
  - [QRCVDTAQ \(Receive from a data queue\)](#)
- Some APIs provide functions not available in CL commands
- APIs have always been available
  - Each new release of IBM i provides new APIs
  - Currently > 1500

© Copyright IBM Corporation 2011

Figure 2-2. What are APIs?

AS106.0

### Notes:

APIs are provided as part of the base support included with IBM i. They are either included as system programs or procedures packaged in service programs which can be called from any high-level language (HLL), including CL.

Many of the APIs have been in use since the days of the System/38. For example:

- **QCMDEXC:** Execute a CL command
- **QCLSCAN:** Scan for a string pattern
- **QCMDCHK:** Check command syntax
- **QSNDATAQ:** Send a data queue message
- **QRCVDTAQ:** Receive a data queue message
- **QCLRDTAQ:** Clear a data queue

## Why use APIs?

IBM i

- Include CL functions in your RPG IV programs
- Access system information
- Access functions not available in CL commands

© Copyright IBM Corporation 2011

Figure 2-3. Why use APIs?

AS106.0

### Notes:

You can use APIs with all supported high-level languages (HLLs) except that the APIs implemented as service programs (\*SRVPGM) can be accessed only by ILE languages.

In some cases, a program (\*PGM) interface is provided so that non-ILE languages can access the function.

Some APIs also require that particular data types and particular parameter passing conventions be used.

IBM i APIs exist in the following environments:

- Original program model (OPM): QXXXXXXX
- Integrated Language Environment (ILE): Qxxxxxxx
- ILE Common Execution Environment (CEE): CEExxxxx
- UNIX-type

APIs complement the familiar command functions, providing additional capabilities with improved performance and flexibility. However, they are not designed for the novice; experience as a programmer is necessary.

Information is supplied to and returned by APIs in the form of parameters. This data is in a form which must be decoded according to strict rules documented in the i Information Center. The Information Center specifies what to do, but not why or how. This is why you must be experienced in using IBM i data types and several other features in order to use APIs effectively.

Special system objects (such as user spaces) are used quite heavily by APIs to deliver information to the programmer, again, in a raw form.

# Categories of APIs

IBM i

- Independent
  - Limited parameter list (QUSCMLDN)
  - Perform a function: Minimal return of data
- Retrieve
  - Fourth character = "R"
  - Return data in parameter **receiver**: Single item
  - Format name required
  - Similar to **RTVxxxx** command
- List
  - Fourth character = "L"
  - Return data in user space (\*USRSPC): Multiple items
  - Format name required
  - Similar to **DSPxxxx** command to \*OUTFILE

© Copyright IBM Corporation 2011

Figure 2-4. Categories of APIs

AS106.0

## Notes:

**Retrieve** and **List** APIs are among the most commonly used APIs. They allow the program to extract information from IBM i in a way similar to the information that can be retrieved by using CL commands.

In many cases, APIs can provide information when no suitable **RTVxxxx** command exists, or when the corresponding **DSPxxxx** command has no OUTFILE.

The format specifies the level of detail required from the API and also defines the structure of the data returned. This must be decoded in the program.

## Example: Using QUSCMDLN

IBM i

- From many programs (for example, SEU), pressing F21 brings up a command window like this one:

```
.....  
:  
:  
: ==> _____  
: F4=Prompt F9=Retrieve F12=Cancel  
:  
:  
.....
```

- You can emulate this in your code
  - Simply issue a call to the QUSCMDLN API
  - It is one of the simplest APIs since it has no parameters

Line 1	Column 1	Replace
000100	D CommandLine	PR ExtPgm('QUSCMDLN')
000200		
000300	/Free	
000400	CallP CommandLine();	
000500	*InLR = *On;	
000600	/End-free	

© Copyright IBM Corporation 2011

Figure 2-5. Example: Using QUSCMDLN

AS106.0

### Notes:

The above program has only one purpose. It displays a command line. You can call the QUSCMDLN API from any program using this code as a guide.

# Communicating with parameters

IBM i

- Input parameters
  - Set to a value before calling the API
- Output parameters
  - Information returned to the calling program by the API
- Both parameters
  - Input and output
  - A structure, containing fields, which are input, output, or both

© Copyright IBM Corporation 2011

Figure 2-6. Communicating with parameters

AS106.0

## Notes:

Each API expects parameters of specific data types and lengths. Each API dictates how each parameter is used; each API may expect input parameters and return output parameters.

Some APIs expect a structure that contains both input and output parameters. You must review the documentation carefully to determine how parameters are passed, and you must pay careful attention to the data types and lengths expected by the particular API that you want to call.

# API parameters: Required, optional, and omissible

IBM i

- Required
  - Must be specified and must be in order required by API
- Optional
  - All or none of the parameters are in an optional group
  - If an optional group is specified, all previous optional groups must be specified
- Omissible
  - Group of parameters, some of which can be omitted by passing a null pointer

© Copyright IBM Corporation 2011

Figure 2-7. API parameters: Required, optional, and omissible

AS106.0

## Notes:

API parameters can be required, optional or omissible.

When you call an API, the protocol for passing parameters is to typically pass an address that points to the information being passed. This method of parameter passing is what we typically do in RPG IV programs.

# API parameter data type considerations

IBM i

In documentation, special considerations:

- **Char(\*)** means:
  - Length not known for certain
  - Length not fixed; to be supplied by program
- **Binary data:**
  - Documented as:
    - **Binary(2)** = 2 bytes signed
    - **Binary(4)** = 4 bytes signed
    - **Binary(4) UNSIGNED** = 4 bytes unsigned
  - RPG IV binary length 4 means precision of 4 digits
  - To define **BINARY(4)** parameter on D-spec:
    - Length of 9 for 4-byte binary
    - Length of 10 for 4-byte integer
- **Note:** Use positional notation on Data definition specification (D-spec) to define precision explicitly

© Copyright IBM Corporation 2011

Figure 2-8. API parameter data type considerations

AS106.0

## Notes:

The description of the parameters that the API requires are intuitive except for cases where:

- **Char(\*)**: This means that the length of the character data is not fixed or known
- **Binary**: The best way to define these parameters is as integers although you find that the members in QSYSINC tend to favor the BINARY data type. Even though the documentation uses binary and from/to notation, we always use integer and length format.

# Example: API parameter definition documentation

IBM i

The screenshot shows the IBM i Information Center interface. The left pane contains a navigation tree under 'Contents' with categories like Programming, Application programming interfaces, API finder, and others. The 'API finder' node is expanded, showing various sub-options. The right pane displays the 'Start Pass-Through (QPASTRPT) API' documentation. It includes a table of required parameters:

	Parameter Description	Type	Default
1	Pass-through information	Input	Char(*)
2	Length of pass-through information	Input	Binary(4)
3	Format name	Input	Char(8)
4	Data	Input	Char(*)
5	Length of data	Input	Binary(4)
6	Error code	I/O	Char(*)

Below the table, it says 'Default Public Authority: \*USE' and 'Threadsafte: No'. A note at the bottom states: 'The Start Pass-Through (QPASTRPT) API starts a 5250 pass-through session and optionally passes up to 1KB of user data from the source system to the target system. This data can be accessed on the target system with the Retrieve Data (QPARTVDA) API.'

Figure 2-9. Example: API parameter definition documentation

AS106.0

## Notes:

Consider a simple API, **QPASTRPT**. Review the parameter definitions necessary to be able to use it.

In order to learn about the parameters, we use the i Information Center at:

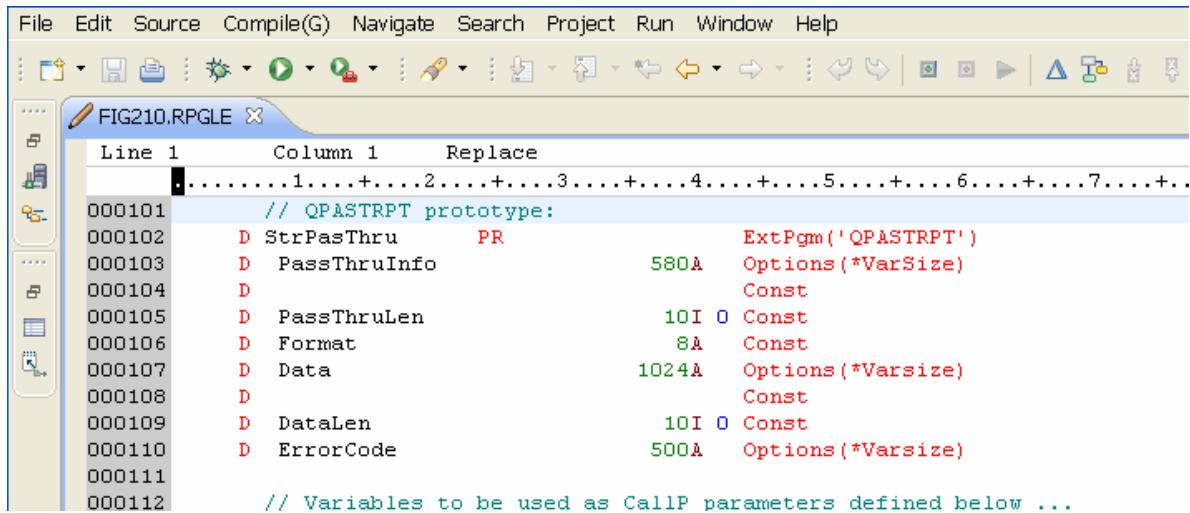
<http://publib.boulder.ibm.com/iseries/v7r1m0>

Once at the Information Center site, expand V7R1, and then:

1. In the left-hand pane, click **Programming**.
2. Click **Application programming interfaces**.
3. Click **API finder**.
4. In the **Find by Name** box, enter *QPASTRPT*, and click **GO**. Notice that you can search APIs by category as well.
5. Click the result (should be a link to QPASTRPT) returned by the API finder.
6. You are presented with the information in the figure above. You can scroll down and review more information specific to the API.

# Example: API parameter definition: Prototype

IBM i



```

File Edit Source Compile(G) Navigate Search Project Run Window Help
.....1....+....2....+....3....+....4....+....5....+....6....+....7....+
000101 // QPASTRPT prototype:
000102 D StrPasThru      PR           ExtPgm('QPASTRPT')
000103 D PassThruInfo          580A Options(*VarSize)
000104 D                                     Const
000105 D PassThruLen          10I 0 Const
000106 D Format               8A  Const
000107 D Data                 1024A Options(*Varsize)
000108 D                                     Const
000109 D DataLen              10I 0 Const
000110 D ErrorCode             500A Options(*Varsize)
000111
000112 // Variables to be used as CallP parameters defined below ...

```

© Copyright IBM Corporation 2011

Figure 2-10. Example: API parameter definition: Prototype

AS106.0

## Notes:

In order to code the prototype for the parameter list, it is necessary to review the documentation for each parameter expected by the API. The API has a number of parameters that are required as you saw in the previous visual:

### Required Parameter Group

#### 1. Pass-through information

- Usage **INPUT**
- Data type and length **CHAR(\*)**

Information associated with establishing the 5250 pass-through session.

#### 2. Length of pass-through information

- Usage **INPUT**
- Data Type and length **BINARY(4) = INTEGER(10, 0)**

The length, in bytes, of the pass-through information parameter. This value must be greater than or equal to 8 and less than or equal to 580.

### 3. Format name

- Usage **INPUT**
- Data type and length **CHAR(8)**

The format name of the pass-through information. The supported format names are:

- PAST0100 Pass-through with up to 10-byte password
- PAST0200 Pass-through with up to 128-byte password

### 4. Data

- Usage **INPUT**
- Data type and length **CHAR(\*)**

User-defined data to be passed to the target system. The format of this data is not defined by the API and is sent to the target system as is.

### 5. Length of data

- Usage **INPUT**
- Data type and length **BINARY(4) = INTEGER(10,0)**

The length of the data parameter.

### 6. Error code

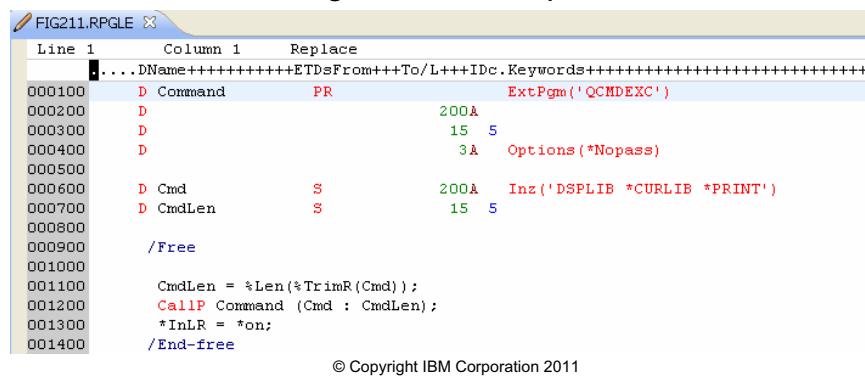
- Usage I/O
- Data type and length **CHAR(\*)**

The structure in which to return error information. We cover the structure for the error code parameter later.

## Example: Using QCMDEXC

IBM i

- Execute command (QCMDEXC) API
  - Allows you to issue a CL command from an RPG IV program
  - RPG IV program constructs and issues the command
  - No need to call a CL program
- Example: A command to display the contents of course library
  - API requires two parameters:
    - First contains the command being issued
    - Second contains the length of the first parameter



```

FIG211.RPGLE
Line 1      Column 1      Replace
      . . . DName++++++ETDsFrom++To/L+++IDc.Keywords+++++
000100  D Command      PR          ExtPgm('QCMDEXC')
000200  D                   200A
000300  D                   15  S
000400  D                   3A  Options(*Nopass)
000500
000600  D Cmd           S          200A  Inz('DSPLIB *CURLIB *PRINT')
000700  D CmdLen        S          15   5
000800
000900  /Free
001000
001100  CmdLen = %Len(%TrimR(Cmd));
001200  CallP Command (Cmd : CmdLen);
001300  *InLR = *on;
001400  /End-free

```

© Copyright IBM Corporation 2011

Figure 2-11. Example: Using QCMDEXC

AS106.0

### Notes:

In this example, we initialize the field **CMD** to the command we want to issue. Then, we use **%Len** and **%TrimR** to determine the length of the *CMD* variable. Use of **%TrimR** is optional as trailing blanks are ignored by the command processor. Note that you can also create the command string in the program using concatenation.

Notice that **QCMDEXC** expects an optional third parameter. This parameter is for the double-byte character set.

An alternate API to QCMDEXC is QCAPCMD. Some think it is a better API than QCMDEXC. While it is more complicated to code, it offers more function including:

- Error handling
- Prompting
- Options for syntax checking only and so on
- Return of changed command string

# Asynchronous communications between jobs

IBM i

- Data queues
  - Use less resources than DB file, message queue, or data area
  - Provide excellent response time
  - Can communicate with more than one job
  - Can include sender ID with message
  - Can communicate with PC users via Client Access or Java
  - Can use DDM to work with data queue (DTAQ) on another iSeries system
- MQSeries
  - Separate software product
  - Cross-platform
  - More open

© Copyright IBM Corporation 2011

Figure 2-12. Asynchronous communications between jobs

AS106.0

## Notes:

Often there is a requirement to communicate between jobs. While using data areas or using message queues is possible, if you want a way that requires less system resources than either of these, you might consider using data queues (object DTAQ). Data queues can be used to pass information between a PC client and IBM i server using IBM i Access or Java.

But, remember, like all queues, only the object description can be saved or restored. You cannot save or restore the contents of a data queue. Data queues should be used for fast communication and information passing. However, use database file for information storage. By default, the contents of a data queue is removed when it is read.

If you need cross-platform messaging, you might consider the MQSeries product.

We discuss the data queue API alternative only.

# Creating data queues

IBM i

Create Data Queue (CRTDTAQ)

Type choices, press Enter.

Data queue . . . . .	DTAQ	<input type="text" value="*CURLIB"/>
Library . . . . .		<input type="text" value="*STD"/>
Type . . . . .	TYPE	<input type="text" value="*NO"/>
Maximum entry length . . . . .	MAXLEN	<input type="text" value="*FIFO"/>
Force to auxiliary storage . . . . .	FORCE	<input type="text" value="*NO"/>
Sequence . . . . .	SEQ	<input type="text" value="KEYLEN"/>
Key length . . . . .	KEYLEN	<input type="text" value="*NO"/>
Include sender ID . . . . .	SENDERID	<input type="text" value="*MAX16MB"/>
Queue size:	SIZE	<input type="text" value="16"/>
Maximum number of entries . . . . .		<input type="text" value="AUTORCL"/>
Initial number of entries . . . . .		<input type="text" value="*NO"/>
Automatic reclaim . . . . .	RMTDTAQ	<input type="text" value="RMTLOCNAME"/>
Remote data queue . . . . .		<input type="text" value="RDB"/>
Library . . . . .		<input type="text" value="More..."/>
Remote location . . . . .		
Relational database . . . . .		

© Copyright IBM Corporation 2011

Figure 2-13. Creating data queues

AS106.0

## Notes:

Before you can access a data queue, you must create it using the **CRTDTAQ** command. These are the main parameters:

- **TYPE:** Value **\*STD** creates a *local* data queue, while **\*DDM** creates a data queue that contains the name of the system where the remote data queue is created, the name of the remote data queue accessed, and the name of the remote (target) system that the data queue is located on.
- **MAXLEN:** Specifies the maximum length of the data to be written to the queue (up to 64512 bytes).
- **FORCE:** Specifies whether the data queue is forced to auxiliary storage when entries are sent or received for this data queue.
- **SEQ:** Specifies that sequence in which entries are written to the queue: first-in-first-out, last-in-first-out, or keyed, where data queue entries are received by key. A key is a prefix added to an entry by its sender.
- **SENDERID:** The userid of the sender is written to the data queue with each message.

- **SIZE**: Specifies the amount of storage allocated for the data queue. This parameter has two elements. The first designates the maximum number of entries, and the second designates the initial number of entries for the data queue. This parameter is valid only when **TYPE(\*STD)** is specified.
- **AUTORCL**: Specifies whether the storage allocated for the data queue is automatically reclaimed (released) when the data queue is empty. This parameter is also valid only when **TYPE(\*STD)** is specified.

You use the **DLTDTAQ** command to delete a data queue.

# Data queue prototypes

```

DTAQPROTO.RPGLE X
Line 1      Column 1      Replace
.....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8.....
000100      * Prototype for API QSNDATAQ - Send To a Data Queue
000200      D SndDtaQ          PR                           EXTPGM('QSNDATAQ')
000300      D DataQueueNam    10A   Const
000400      D DataQueueLib    10A   Const
000500      D DataLength       5P 0 Const
000600      D DataBuffer       32767A  Const Options(*Varsize)
000700      * Optional parameter group (Keyed DTAQ)
000800      D KeyLength        3P 0 Const Options(*Nopass)
000900      D KeyBuffer        256A  Const Options(*Nopass : *Varsize)
001000      D AsyncRqs         10A   Const Options(*Nopass : *Varsize)
001100      D DataFrmJrn       10A   Const Options(*Nopass)
001200      *
001300      * Prototype for API QRCVDTAQ - Received From a Data Queue
001400      D RcvDtaQ          PR                           EXTPGM('QRCVDTAQ')
001500      D DataQueueNam    10A   Const
001600      D DataQueueLib    10A   Const
001700      D DataLength       5P 0
001800      D DataBuffer       32767A           Options(*Varsize)
001900      D WaitTime         5P 0 Const
002000      * Optional parameter group 1 (Keyed DTAQ)
002100      D KeyOrder         2A   Const Options(*Nopass)
002200      D KeyLength        3P 0 Const Options(*Nopass)
002300      D KeyBuffer        256A           Options(*Nopass : *Varsize)
002400      D SndLength        3P 0 Const Options(*Nopass)
002500      D SndBuffer        44A   Options(*Nopass : *Varsize)
002600      * Optional parameter group 2
002700      D RemoveMsg        10A   Const Options(*Nopass : *Omit)
002800      D RcvSize          5P 0 Const Options(*Nopass : *Omit)
002900      D Error            32767A  Options(*Nopass : *Varsize)

```

© Copyright IBM Corporation 2011

Figure 2-14. Data queue prototypes

AS106.0

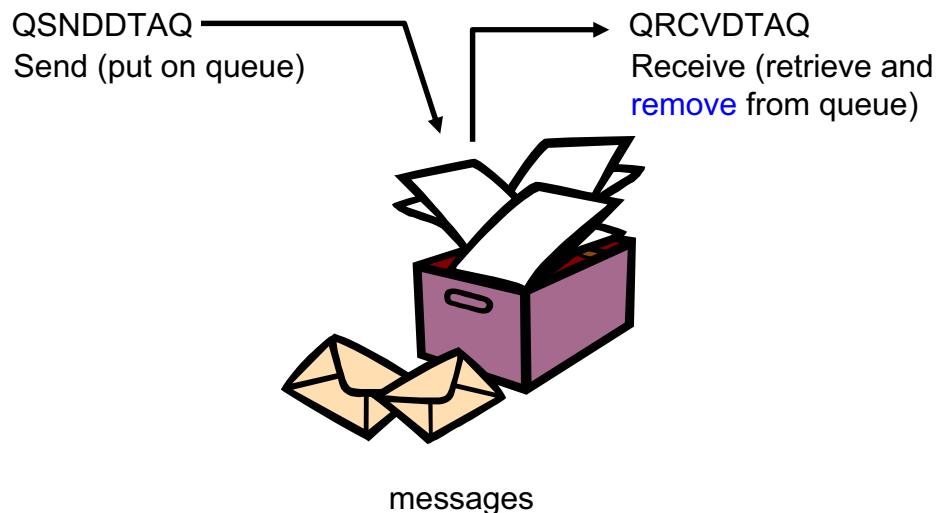
## Notes:

The data queue APIs require you to pass parameters in order to communicate with them. We are going to look at two APIs:

1. QSNDATAQ: Sends data to a data queue specified as one of the parameters. Distributed data management (DDM) data queues are supported using this API. This means that you can use this API to send data to a data queue that exists on a remote i system.
2. QRCVDTAQ: Receives data from a data queue specified as a parameter. Distributed data management (DDM) data queues are supported using this API. This means that you can use this API to receive a message from a data queue that exists on a remote i system. However, using this API to receive messages without removing them from the data queue is not supported for DDM data queues.

# Using QSNDTAQ/QRCVDTAQ

IBM i



\*DTAQ object

Create: **CRTDTAQ** command

Delete: **DLTDTAQ** command

© Copyright IBM Corporation 2011

Figure 2-15. Using QSNDTAQ/QRCVDTAQ

AS106.0

## Notes:

## Example: Using QSNDATAQ/QRCVDTAQ (1 of 3)

IBM i

```

DTAQSR.RPGL
Line 1      Column 1      Replace
000100    D/Copy APISRC,DTAQPROTO
000200    // Program variable definitions
000300    D Length      S          5P 0
000400    D Data        S          40A
000500    D KeyBuf     S          6A
000600    D Sender     S          44A
000700
000800    D WaitTime    C          -1
000900    // Receives message from client
001000    /FREE
001100    RcvDtaQ('DTAQFIFO' : '*CURLIB'
001200      : Length : Data : WaitTime
001300      : *Blank : *Zero : KeyBuf
001400      : *Size(Sender) : Sender);
001500    // Sends answer to client
001600    Data = 'Hello Client '
001700      + %Subst(Sender : 29 : 6)
001800      + ', thanks for calling';
001900
002000    SndDtaQ('DTAQKEYED' : '*CURLIB'
002100      : %Len(%Trim(Data)) : Data
002200      : 6 : %Subst(Sender : 29 : 6));
002300
002400    *InLR = *On;
002500  /END-FREE

```

© Copyright IBM Corporation 2011

Figure 2-16. Example: Using QSNDATAQ/QRCVDTAQ (1 of 3)

AS106.0

### Notes:

We have two programs that use data queues to communicate with one another, a *client* program and a *server* program. The client program writes a message to a data queue. The server program monitors the data queue and replies to the client.

Shown in this visual is the *server* program. It must be able to be started and then be able to wait for the client (or any other program) to place an entry in a data queue that it monitors. Some things to notice are:

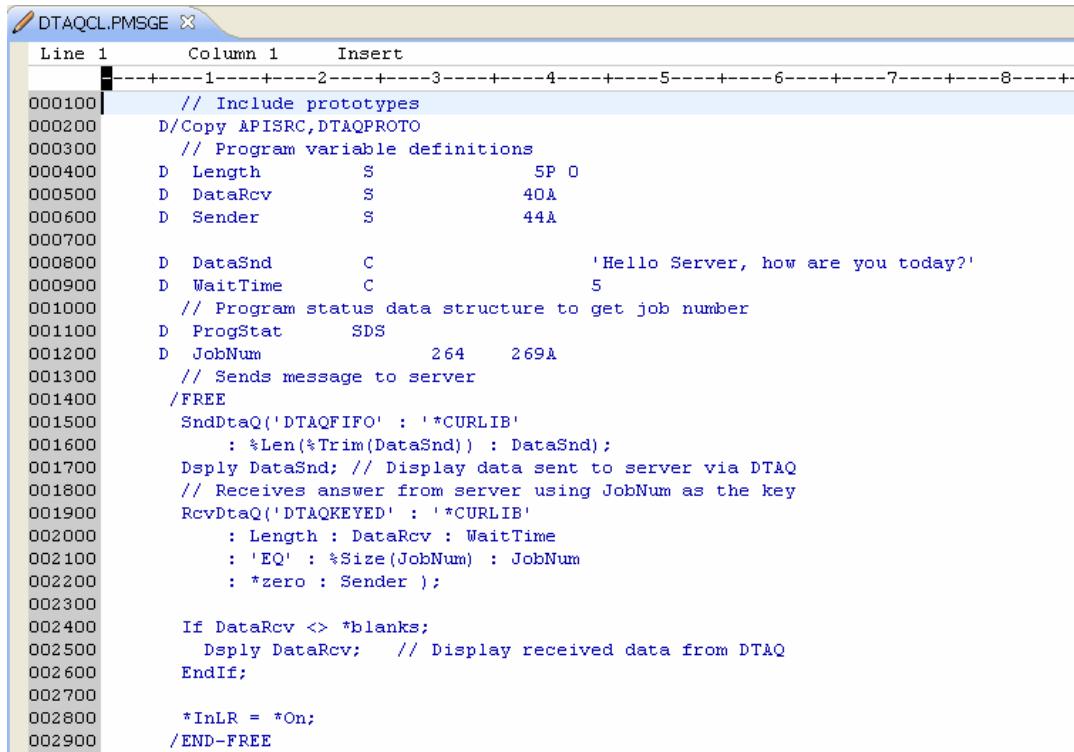
1. The server has the wait time parameter set to **-1**. This means that the API which retrieves messages from the data queue waits indefinitely for an entry to be written to the data queue specified in the call.
2. Since our server program must wait for the client to do something to a data queue, we need to run the server program as a batch job and the client as an interactive job.
3. The server program retrieves messages from a client, picking up the data written (40 characters) and the sender information (information that is a combination of information

built from the job name, user profile name, job number, and the sender's current user profile name).

4. The server can receive messages from many clients and writes replies to a keyed data queue using the job number portion of the sender parameter as a key.

## Example: Using QSNDTQAQ/QRCVDTAQ (2 of 3)

IBM i



```

DTAQCL.PMSGE X
Line 1      Column 1      Insert
-----+---1---+---2---+---3---+---4---+---5---+---6---+---7---+---8---+
000100 // Include prototypes
000200 D/Copy APISRC,DTAQPROTO
000300 // Program variable definitions
000400 D Length           S          5P 0
000500 D DataRcv          S          40A
000600 D Sender           S          44A
000700
000800 D DataSnd          C          'Hello Server, how are you today?' 
000900 D WaitTime          C          5
001000 // Program status data structure to get job number
001100 D ProgStat          SDS
001200 D JobNum            264     269A
001300 // Sends message to server
001400 /FREE
001500   SndDtaQ('DTAQFIFO' : '*CURLIB'
001600     : %Len(%Trim(DataSnd)) : DataSnd);
001700   Dsply DataSnd; // Display data sent to server via DTAQ
001800 // Receives answer from server using JobNum as the key
001900   RcvDtaQ('DTAQKEYED' : '*CURLIB'
002000     : Length : DataRcv : WaitTime
002100     : 'EQ' : %Size(JobNum) : JobNum
002200     : *zero : Sender );
002300
002400   If DataRcv <> *blanks;
002500     Dsply DataRcv; // Display received data from DTAQ
002600   EndIf;
002700
002800   *InLR = *On;
002900 /END-FREE

```

© Copyright IBM Corporation 2011

Figure 2-17. Example: Using QSNDTQAQ/QRCVDTAQ (2 of 3)

AS106.0

### Notes:

QSNDTQAQ is the client program that sends a message to the server program by writing the message to a data queue. The client uses QSNDTQAQ to write a message to the queue being monitored by the server program. Some key points:

1. The message '*Hello, server. How are you today?*' is written to the data queue using the QSNDTQAQ API.
2. The client waits 5 seconds (WaitTime) and retrieves the response (using QRCVDTAQ) written to the keyed data queue by the server program. The client uses its job number (using the Program Status data structure) as a key to the queue in order to fetch information specifically for it (and not for any other client).
3. The client then displays the retrieved message and ends.

## Example: Using QSNDATAQ/QRCVDTAQ (3 of 3)

IBM i

- Create data queues:
  - To send data from the client to the server:
    - **CRTDTAQ DTAQ(DTAQFIFO) MAXLEN(40) SENDERID(\*YES)**
  - To send data from the server back to the client:
    - **CRTDTAQ DTAQ(DTAQKEYED) MAXLEN(40) SEQ(\*KEYED) KEYLEN(6) SENDERID(\*NO)**
- **SBMJOB CMD(CALL PGM(DTAQSR))**
  - Server waits to be called by client
- Call **DTAQCL**
  - Sends message to server '*Hello Server, how are you today?*'
  - Server replies '*Hello Client 004248, thanks for calling*'

© Copyright IBM Corporation 2011

Figure 2-18. Example: Using QSNDATAQ/QRCVDTAQ (3 of 3)

AS106.0

### Notes:

This visual lists all the steps necessary to write and run this sample application.

1. Create Data Queues: We need to create two data queues. One is used by client programs to send messages to the server program that monitors this queue. Since the server handles messages as they are received, we create a first-in-first-out (FIFO) data queue. The server knows which client sent the message because we included sender information (SENDERID).

The second data queue is a keyed data queue. The server program writes replies to any client to this queue using the client's job number as the key. This job number is part of the sender information.

2. The server program, **DTAQSR**, is started as a batch job. It waits indefinitely until a client sends it a message. Once the queue **DTAQFIFO** is empty, it ends.
3. The client, **DTAQCL**, is started interactively. This program sends a message to the server '*Hello Server, how are you today?*' using the **DTAQFIFO**. The server replies '*Hello Client 004248, thanks for calling*' to the **DTAQKEYED**,

using the client's job number as a key for the message. This message is retrieved by the client program.

## Example: Using QMHSNDPM

IBM i

- Send a message from an RPG program
- Parameters similar to **SNDPGMMMSG CL command**

```

FIG219.RPGL X
Line 1      Column 1      Replace
.....DName++++++ETDsFrom++To/L+++IDc.Keywords+++++=====
000201    D ErrorMsgId   S          7A
000202    D MsgFile       S          20A
000300    D MsgData       S          40A
000400    D MsgDtaLen    S          10I 0 Inz(%Len(MsgData))
000500 >>1  D MsgType      S          10A  Inz('*INFO')
000600 >>2  D CallStack     S          10A  Inz('**')
000700 >>3  D CallStackCtr  S          10I 0 Inz(2)
000800    D MsgKey       S          4A
000900    D ErrorCode     S          6A
001000
001100 >>4  D SendPgmMsg   PR          ExtPgm('QMHSNDPM')
001200    D               S          7A
001300    D               S          20A
001400    D               S          40A
001500    D               S          10I 0
001600    D               S          10A
001700    D               S          10A
001800    D               S          10I 0
001900    D               S          4A
002000    D               S          6A
002100
002200    /Free
002201 >>5  Msgdata = 'Message from SNDMSG to Class AS10/S6199';
002300 >>6  CallP SendPgmMsg (ErrorMsgId : MsgFile : MsgData : MsgDtaLen : MsgType:
002400           CallStack : CallStackCtr : MsgKey : ErrorCode);
002500 *InLR = *On;
002600 /End-free

```

© Copyright IBM Corporation 2011

Figure 2-19. Example: Using QMHSNDPM

AS106.0

### Notes:

The QMHSNDPM API enables you to send a message from your program. The message itself can be of any type, and what is sent is determined by various parameters that you send to the API.

If you send an escape message, the program that issues the message automatically ends and returns to the calling program.

The call stack count determines to which location in the call stack the message is sent. A value of **0** (zero) sends the message to the message queue of the entry specified by the call stack entry parameter. A value of **1** (one) sends the message to the program that is one earlier in the call stack. Other values for call stack count will send the message higher up the call stack. The main parameters are:

- **ERRORMSGID** and **MSGFILE** specify the message identifier and its message file when a message from a message file is to be sent.
- **MSGDATA** contains any variable data for the message, and **MSGDTALEN** must indicate the length of **MSGDATA**.

- **MSGTYPE (1)** indicates the message type. In this instance, **\*INFO** indicates an information message.
- The **CALLSTACK** and **CALLSTACKCTR** parameters indicate where the message is to be sent, with the current value (a **CALLSTACK** of **\***) pointing to the stack **(2)** in which the program is running. Messages will appear at the bottom of your display and are written to the job log when **CALLSTACKCTR = 2 (3)**. A **CALLSTACKCTR** of 0 will write to the job log; messages will not be displayed at the bottom of your display.
- The **MSGKEY** and **ERRORCODE** parameters are not set but are required by the API.

Other points of interest:

- The Prototype **(4)** for the QMHSNDPM is a **\*PGM** object and so we must code the **EXTPGM** parameter. The prototype would normally be a copy member.
- The message is built as a program variable, **MSGDATA (5)**.
- The API is called through **CALLP (6)**.

The QMHSNDPM API is documented in the i Information Center.

## Machine exercise: Using system APIs (1 of 2)

IBM i



© Copyright IBM Corporation 2011

Figure 2-20. Machine exercise: Using system APIs (1 of 2)

AS106.0

### Notes:

Perform the machine exercise *using system APIs 1*.

## Retrieve APIs

IBM i

- Fourth letter of name = R
- Retrieve information about a single thing
- Return the information through the parameter list
- Returned information placed in receiver variable, usually defined as **CHAR(\*)**
- Receiver variable usually contains embedded variables
- Data structure is the perfect method in RPG for defining receiver variables
- Layout of receiver also found in QSYSINC member

© Copyright IBM Corporation 2011

Figure 2-21. Retrieve APIs

AS106.0

### Notes:

## Format parameter of retrieve and list APIs

IBM i

- Tells API what data to retrieve
- Describes level of detail desired
- Defines how data is presented to caller (strict layout)
- Layouts described in IBM i Information Center
- Code included in QSYSINC

© Copyright IBM Corporation 2011

Figure 2-22. Format parameter of retrieve and list APIs

AS106.0

### Notes:

The 8-character format name must be supplied as a parameter to retrieve and list APIs. It tells the API what level of detail to supply back to the calling program and dictates how this data is structured.

For example, when using the QUSROBJD (Retrieve Object Description) API, four distinct data formats are available:

1. *OBJD0100* provides basic information (fastest)
2. *OBJD0200* provides information similar to that displayed by Programming Development Manager (PDM)
3. *OBJD0300* provides service information
4. *OBJD0400* provides full object information (slowest)

The layout of these structures is documented in the i Information Center. For the C, RPG, and COBOL languages, source code copy members are provided in library QSYSINC to simplify the coding effort.

# QSYSINC library

IBM i

- OPM APIs
  - QRPGSRC file: Member name is OPM API program name or program name with E replacing Q for members containing array definitions
  - QRPGLESRC file: Member is OPM API program name
- ILE APIs
  - QRPGLESRC file: Member name is service program name or API program name
- Variable length parameters included as comments in members
  - Uncomment and edit the lengths
- QUSGEN: Defines the generic header for list APIs
- QUSEC: Contains the structure for the error code

© Copyright IBM Corporation 2011

Figure 2-23. QSYSINC library

AS106.0

## Notes:

QSYSINC (system includes) is a library included with IBM i that you optionally install. For IBM i Version 7, you should install IBM i option 13, **System Openness Includes**.

The QSYSINC library provides source include (copy) files shipped with the IBM i APIs. The include files define only the fixed portion of the formats for API parameters.

The entire member *should not* be copied into your source member using /COPY or /INCLUDE:

- Each member contains all definitions relevant to the API. Only a subset, for example, a particular format, is required by your program at any one time.
- The members may change at any time in the future, affecting subsequent program compilations.
- Use Copy/Paste to import only the lines of source code required. You could use Source Entry Utility (SEU) Browse and Copy (F15) or open another member in RSE/LPEX and use Copy/Paste.

- The naming conventions and coding style may not suit your installation standards. You can devise your own copy members or use external file definitions.

**Note:** Not all APIs are included as members in QSYSINC.

# Retrieve API example: USROBJD API parameters

IBM i

Name	Use	Type
Receiver variable	Output	Char(*)
Length of receiver variable	Input	Binary(4)
Format name	Input	Char(8)
Qualified object and library name	Input	Char(20)
Object type	Input	Char(10)
Error code (optional)	I/O	Char(*)

Determines size of

© Copyright IBM Corporation 2011

Figure 2-24. Retrieve API example: USROBJD API parameters

AS106.0

## Notes:

The above parameters are entirely character or 4-byte binary.

RPG IV allows the I and U data types to be used for binary. For instance, a **Binary(4)** value can be defined as a 10-byte integer. They are labeled input, output, and both in relation to the API.

Some parameters may be optional and can be omitted.

It is usual to find a variable length character parameter, **Char(\*)** with an associated 4-byte binary parameter. The latter tells the API how long the character variable is, a similar approach to the QCMDEXC API.

## Retrieve object description (QUSROBJD)

IBM i

- Like **DSPOBJD** command
- Use the QUSROBJD API to:
  - Determine who owns which objects in specific library
  - Provide disk management functions based on the object's size and use
  - Perform analysis based on when object was last saved or last updated
  - Check whether a source member was used to create specific object
  - Work with list of objects created by QUSLOBJ API

© Copyright IBM Corporation 2011

Figure 2-25. Retrieve object description (QUSROBJD)

AS106.0

### Notes:

This API is similar to **DSPOBJD** but it returns information to your program. You can select to have the information returned in one of three formats, depending upon the level of detail that you require.

The kind of information you can retrieve includes the ID of the object's owner, the date the object was created, and the date it was last changed.

The parameters passed to the API include the name of the object, the library in which it is located, and the object's type.

## Example: QUSROBJD

IBM i

```

000100 // Prototype for API QUSROBJD - Retreive Object Description
000200 D RtvObjD      Pr          ExtPgm('QUSROBJD')
000300 D ObjDta         90A
000400 D ObjDtaLen      10I 0 Const
000500 D FormatName     8A   Const
000600 D ObjectLib      20A   Const
000700 D ObjectType     10A   Const
000800 D Error          16A   Options(*NOPASS)
000900 // Format OBJDO100 with received fields
001000 D QUSD01DS        DS
001100 D DataRet         10I 0
001200 D DataAvail       10I 0
001300 D ObjectName      10A
001400 D LibName         10A
001500 D ObjectType      10A
001600 D ReturnLib       10A
001700 D AuXstorPool    10I 0
001800 D ObjectOwn       10A
001900 D ObjectDomain    2A
002000 D CrtDatTime     13A
002100 D ChgDatTime     13A
002200 // Program variable definitions
002300 D ErrorCode        DS
002400 D BytesProv       10I 0
002401 D BytesAvail      10I 0
002402 D ExceptID        7A
002403 D Reserved         1A
002404
002500 /Free
002600 // Call QUSROBJD to retrieve description of object
002700 CallP RtvObjD(QUSD01DS : %Len(QUSD01DS) :
002800     'OBJDO100' : 'QRPGLESRC AS10XXX' : 'FILE' :
002900     ErrorCode);
003000 Dsply ObjectOwn ;
003100 Dsply CrtDatTime;
003200 Dsply ChgDatTime;
003300
003400 // ObjectOwn = 'RJSLANEY'
003500 // CrtDatTime = '1020515090255' (May. 15, 2002 09:02:55)
003600 // ChgDatTime = '1031008085404' (Oct. 08, 2003 08:54:04)
003700
003800 *InLR = *On;
003900 /End-free

```

© Copyright IBM Corporation 2011

Figure 2-26. Example: QUSROBJD

AS106.0

### Notes:

This sample program retrieves and displays the owner of an RPG IV source file in a library. We also retrieve the date the object was located and the last date that it was modified.

## Common problems and error handling

IBM i

- Common problems when dealing with APIs
  - Incorrect parameter definition
  - Variables declared incorrectly (for example, 4B0 instead of 10I0)
  - Data structures not correctly defined/aligned
- Error handling options
  - RPG e-Extender on CALLP
  - Use MONITOR operation
  - Take advantage of API error code as catch-all

© Copyright IBM Corporation 2011

---

Figure 2-27. Common problems and error handling

AS106.0

### Notes:

Be aware of some of the common mistakes you can make and the problems they can cause.

# Error handling DS: QUSEC (1 of 2)

IBM i

```

Line 38      Column 15      Replace      Browse
.....1....#....2....+....3....+....4....+....5....+....6....+....7....+....8.
000038      D**** END HEADER FILE SPECIFICATIONS ****
000039      D***** Record structure for Error Code Parameter
000040      D*Record structure for Error Code Parameter
000041      D*****
000042      D*NOTE: The following type definition only defines the fixed
000043      D* portion of the format. Varying length field Exception
000044      D* Data will not be defined here.
000045      D***** DS
000046      DQUSEC
000047      D*
000048      D QUSBPRV           1      4B 0          Qus EC
000049      D*
000050      D QUSBAVL            5      8B 0          Bytes Provided
000051      D*
000052      D QUSEI              9      15           Bytes Available
000053      D*
000054      D QUSERVED           16     16           Exception Id
000055      D*
000056      D*QUSED01           17     17           Reserved
000057      D*
000058      D*                      Varying length

```

© Copyright IBM Corporation 2011

Figure 2-28. Error handling DS: QUSEC (1 of 2)

AS106.0

## Notes:

An API error code parameter is common to all of the system APIs. OS/400 or IBM i APIs include an error code parameter to return error codes and exception data to the application. The error code parameter is a variable-length structure that contains the information associated with an error condition. The error code parameter can be one of two variable-length structures, format **ERRC0100** or format **ERRC0200**.

For some APIs, the error code parameter is optional. If you do not code the optional error code parameter, the API returns diagnostic and escape messages. If you do code the optional error code parameter, the API returns only escape messages or error codes; it never returns diagnostic messages.

## Error handling DS: QUSEC (2 of 2)

IBM i

### Format ERRC0100

Offset		Use	Type	Field
Dec	Hex			
0	0	INPUT	BINARY(4)	Bytes provided
4	4	OUTPUT	BINARY(4)	Bytes available
8	8	OUTPUT	CHAR(7)	Exception ID
15	F	OUTPUT	CHAR(1)	Reserved
16	10	OUTPUT	CHAR(*)	Exception data

### Format ERRC0200

Offset		Use	Type	Field
Dec	Hex			
0	0	INPUT	BINARY(4)	Key
4	4	INPUT	BINARY(4)	Bytes provided
8	8	OUTPUT	BINARY(4)	Bytes available
12	C	OUTPUT	CHAR(7)	Exception ID
19	13	OUTPUT	CHAR(1)	Reserved
20	14	OUTPUT	BINARY(4)	CCSID of the CCHAR data
24	18	OUTPUT	BINARY(4)	Offset to the exception data
28	1C	OUTPUT	BINARY(4)	Length of the exception data
		OUTPUT	CHAR(*)	Exception data

© Copyright IBM Corporation 2011

Figure 2-29. Error handling DS: QUSEC (2 of 2)

AS106.0

### Notes:

The most commonly used format is *ERRC0100*. One field in that structure is an INPUT field; it controls whether an exception is returned to the application or the error code structure is filled in with the exception information. When the bytes provided field is greater than or equal to 8, the rest of the error code structure is filled in with the OUTPUT exception information associated with the error. When the bytes provided INPUT field is zero, all other fields are ignored and an exception is returned.

Format *ERRC0200* must be used if the API caller wants convertible character (CCHAR) support. Format ERRC0200 contains two INPUT fields, **Key** and **Bytes provided**. The first field, called the **Key** field, must contain a -1 to use CCHAR support. When the **Bytes provided** field is greater than or equal to 12, the rest of the error code structure is filled in with the OUTPUT exception information associated with the error. When the **Bytes provided** INPUT field is zero, all other fields are ignored and an exception is returned.

**Bytes provided** is an input parameter which controls whether an exception is returned to the application or the error code structure is filled in with the exception information.

Consider these points:

- If this field is 0, all other fields are ignored and an exception is returned.
- If the value is equal to or greater than 8, the rest of the error code structure is filled in with the exception information associated with the error, and no exception is returned.

**Bytes available** is the length of the error information returned from the API. If this is 0, no error was detected and none of the fields that follow this field in the structure are changed.

**Bytes provided** is the number of bytes that the calling application provides for the error code. If the API caller is using format ERRC0100, the **Bytes provided** must be 0, 8, or more than 8. If more than 32,783 bytes (32 KB for exception data plus 16 bytes for other fields) are specified, it is not an error, but only 32,767 bytes (32 KB) can be returned in the exception data.

Most programs use this format. Notice that the difference in using the two formats is based upon the value you place in **Bytes provided** as well as the **Key** field.

If the API caller is using format ERRC0200, the bytes provided must be 0, 12, or more than 12. If more than 32,799 bytes (32 KB for exception data plus 32 bytes for other fields) are specified, it is not an error, but only 32,767 bytes (32 KB) can be returned in the exception data.

**0:** If an error occurs, an exception is returned to the application to indicate that the requested function failed.

**>=8:** If an error occurs, the space is filled in with the exception information. No exception is returned. This only occurs if format ERRC0100 is used.

**>=12:** If an error occurs, the space is filled in with the exception information. No exception is returned. This only occurs if format ERRC0200 is used.

**CCSID of the CCHAR data** is the coded character set identifier (CCSID) of the convertible character (CCHAR) portion of the exception data. The default is **0**.

**0:** The default job CCSID.

**CCSID:** A valid CCSID number. The valid CCSID range is 1 through 65535, but not 65534.

**Exception ID** field contains the message identifier for the error condition. Exception data is a variable-length character field, which contains the insert data associated with the exception ID.

**Key** is the key value that enables the message handler error function if CCHAR support is used. This value should be -1 if CCHAR support is expected.

**Length of the exception data** contains the length, in bytes, of the exception data returned in the error code.

**Offset to the exception data** is the offset from the beginning of the error code structure to the exception data in the error code structure.

**Reserved** contains a 1-byte reserved field.

## Other APIs

IBM i

- List APIs
  - Like DSPxxx commands
- CEE APIs
  - Bindable ILE APIs

© Copyright IBM Corporation 2011

Figure 2-30. Other APIs

AS106.0

### Notes:

These APIs are similar in function to the DSPxxx commands. Some examples:

- List Database File Members (QUSLMBR)
- List Database Relations (QDBLDBR)
- List Fields (QUSLFLD)
- List Object Locks (QWCLOBJL)
- List Objects (QUSLOBJ)

These APIs use a special object called a *user space* to hold the information output by the API.

The first step is to create a user space. Next, we tell you what an user space is and then explain how to create and use it in your applications.

## Machine exercise: Using system APIs (2 of 2)

IBM i



© Copyright IBM Corporation 2011

Figure 2-31. Machine exercise: Using a system APIs (2 of 2)

AS106.0

### Notes:

Perform the machine exercise *using system API 2*.

## Checkpoint (1 of 2)

IBM i

1. Application programming interface (API) programs
  - a. Can utilize and return information in the form of parameters
  - b. Have always been available
  - c. Offer improved performance and flexibility over familiar CL commands
  - d. All of the above
2. In the API parameter tables, **Char(\*)** represents
  - a. Character data
  - b. Graphics data
  - c. Character data of unknown length
  - d. Double-byte character set (DBCS) data
3. The format parameter of retrieve and list APIs
  - a. Describe the level of detail desired
  - b. Determine the layout of the data returned to the caller
  - c. Are described in the i Information Center
  - d. All of the above

© Copyright IBM Corporation 2011

---

Figure 2-32. Checkpoint (1 of 2)

AS106.0

### Notes:

## Checkpoint (2 of 2)

IBM i

4. True or False: An 8-character format name must be supplied as a parameter to retrieve APIs.
  
5. The \_\_\_\_\_ library on the i system can be used to code the data structure (DS) for the format of the returned data and the prototype for an API.
  - a. QSYS
  - b. QGPL
  - c. QSYSINC
  - d. QRPGLE

© Copyright IBM Corporation 2011

Figure 2-33. Checkpoint (2 of 2)

AS106.0

### Notes:

## Unit summary

IBM i

Having completed this unit, you should be able to:

- Find API documentation in the i Information Center
- Determine parameters and prototype definition necessary to call a system API
- Code RPG IV programs that call system APIs

© Copyright IBM Corporation 2011

Figure 2-34. Unit summary

AS106.0

### **Notes:**

# Unit 4. RPG IV features

## What this unit is about

This unit covers many features that can be used to make your program much more functional. Some are used more than others. Many programmers are unaware that these features are available with the compiler.

## What you should be able to do

After completing this unit, you should be able to:

- Code RPG IV programs that include conditional directives
- Describe user spaces and how to access them using an API
- Describe the purpose of the pointer data type
- Use a pointer to access a data item in an RPG IV program
- Create storage outside an RPG IV program dynamically
- Use a pointer to access storage outside of your RPG IV program

## Accountability

- Machine Exercise
- Checkpoint questions

## Unit objectives

IBM i

After completing this unit, you should be able to:

- Code RPG IV programs that include conditional directives
- Describe user spaces and how to access them using an API
- Describe the purpose of the pointer data type
- Use a pointer to access a data item in an RPG IV program
- Create storage outside an RPG IV program dynamically
- Use a pointer to access storage outside of your RPG IV program

© Copyright IBM Corporation 2011

Figure 3-1. Unit objectives

AS106.0

### Notes:

## 4.1. Compiler directives

# Compiler directives to control appearance of compile listing

IBM i

```
/TITLE My Title Skip to a new page and print a title  
/EJECT Skip to a new page  
/SPACE nnn Space nnn lines
```

© Copyright IBM Corporation 2011

Figure 3-2. Compiler directives to control appearance of compile listing

AS106.0

## Notes:

The compiler directive statements /TITLE, /EJECT, and /SPACE allow you to specify heading information for the compiler listing and to control the spacing of the compiler listing.

### /TITLE

Positions	Entry
-----------	-------

7-12	/TITLE
------	--------

13	Blank
----	-------

14-100	Title information
--------	-------------------

A program can contain more than one /TITLE statement. Each /TITLE statement provides heading information for the compiler listing until another /TITLE statement is encountered. A /TITLE statement must be the first RPG specification encountered to print information on the first page of the compiler listing. The information specified by the /TITLE statement is printed in addition to compiler heading information.

The /TITLE statement causes a skip to the next page before the title is printed. The /TITLE statement is not printed on the compiler listing.

### /EJECT

Positions	Entry
7-12	/EJECT
13-49	Blank
50-100	Comments

If the spool file is already at the top of a new page, /EJECT does not advance to a new page. /EJECT is not printed on the compiler listing.

### /SPACE

Use the compiler directive /SPACE to control line spacing within the source section of the compiler listing. The following entries are used for /SPACE:

Positions	Entry
7-12	/SPACE
13	Blank
14-16	A positive integer value from 1 through 112 that defines the number of lines to space on the compiler listing. The number must be left-adjusted.
17-49	Blank
50-100	Comments

If the number specified in positions 14 through 16 is greater than 112, 112 is used as the /SPACE value. If the number specified in positions 14 through 16 is greater than the number of lines remaining on the current page, subsequent specifications begin at the top of the next page.

/SPACE is not printed on the compiler listing but is replaced by the specified line spacing. The line spacing caused by /SPACE is in addition to the two lines that are skipped between specification types.

# Compiler directive to insert code from another source member

IBM i

```
/COPY libraryname/filename,membername
```

1. C/COPY MBR1

2. I/COPY SRCFIL,MBR2

3. O/COPY SRCLIB/SRCFIL,MBR3

4. O/COPY "SRCLIB!"/"SRC>3","MBR>3"

© Copyright IBM Corporation 2011

Figure 3-3. Compiler directive to insert code from another source member

AS106.0

## Notes:

The /COPY compiler directive causes records from other files to be inserted, at the point where the /COPY occurs with the file being compiled. The inserted files may contain any valid specification including /COPY up to the maximum nesting depth specified by the COPYNEST keyword (32 when not specified).

The /COPY statement is entered in the following way:

<b>Positions</b>	<b>Entry</b>
------------------	--------------

7-11            /COPY

12              Blank

13-49          Identifies the location of the member to be copied (merged). The format is *libraryname/filename,membername*.

- A member name must be specified.
- If a file name is not specified, QRPGLESRC is assumed.

- If a library is not specified, the library list is searched for the file. All occurrences of the specified source file in the library list are searched for the member until it is located or the search is complete.
- If a library is specified, a file name must also be specified.

## 50-100              Comments

To facilitate application maintenance, you may want to place the prototypes of exported procedures in a /COPY member. If you do, be sure to place a /COPY directive for that member in both the module containing the exported procedure and any modules that contain calls to the exported procedure.

During compilation, the specified file members are merged into the program at the point where the /COPY statement occurs. All /COPY members appear in the COPY member table.

Nesting of /COPY directives is allowed. A /COPY member may contain one or more /COPY directives (which in turn may contain further /COPY directives and so on). The maximum depth to which nesting can occur can be set using the COPYNEST control specification keyword. The default maximum depth is 32.

You must ensure that your nested /COPY files do not include each other infinitely. Use conditional compilation directives at the beginning of your /COPY files to prevent the source lines from being used more than once.

Explanation of examples in the visual:

1. Copies from member MBR1 in source file QRPGLESRC. The current library list is used to search for file QRPGLESRC.
2. Copies from member MBR2 in file SRCFIL. The current library list is used to search for file SRCFIL. Note that the comma is used to separate the file name from the member name.
3. Copies from member MBR3 in file SRCFIL in library SRCLIB.
4. Copies from member *MBR>3* in file *SRC>3* in library SRCLIB.

# Compiler directives to control which source statements are compiled

IBM i

- Conditionally include or exclude sections of source code.
- Control which lines are read: /IF, /ELSEIF, /ELSE, and /ENDIF
- Condition expressions for /IF groups
- DEFINED(*condition-name*)
  - NOT DEFINED(*condition-name*)
- Define list of active, named conditions: /DEFINE and /UNDEFINE
- Skip the rest of the source: /EOF

© Copyright IBM Corporation 2011

Figure 3-4. Compiler directives to control which source statements are compiled

AS106.0

## Notes:

The conditional compilation directive statements /DEFINE, /UNDEFINE, /IF, /ELSEIF, /ELSE, /ENDIF, and /EOF allow you to select or omit source records. The compiler directive statements must precede any compile-time array or table records, translation records, and alternate collating sequence records (that is, \*\* records).

Listed below are guidelines to remember when using conditional directives:

- Directive statements allow you to conditionally include or exclude sections of source code from the compile.
- Condition names can be added or removed from a list of currently defined conditions using the defining condition directives /DEFINE and /UNDEFINE.
- Condition expressions DEFINED(*condition-name*) and NOT DEFINED(*condition-name*) are used within testing condition /IF groups.
- Testing condition directives /IF, /ELSEIF, /ELSE and /ENDIF control which source lines are to be processed by the compiler.

- The /EOF directive tells the compiler to ignore the rest of the source lines in the current source member.

### **/IF condition-expression (Positions 7-9)**

The /IF compiler directive is used to test a condition expression for conditional compilation. The following entries are used for /IF:

<b>Positions</b>	<b>Entry</b>
7 - 9	/IF
10	Blank
11 - 80	Condition expression
81 - 100	Comments

If the condition expression is true, source lines following the /IF directive are selected to be read by the compiler. Otherwise, lines are excluded until the next /ELSEIF, /ELSE, or /ENDIF in the same /IF group.

### **/ELSEIF condition-expression (Positions 7-13)**

The /ELSEIF compiler directive is used to test a condition expression within an /IF or /ELSEIF group. The following entries are used for /ELSEIF:

<b>Positions</b>	<b>Entry</b>
7 - 13	/ELSEIF
14	Blank
15 - 80	Condition expression
81 - 100	Comments

If the previous /IF or /ELSEIF was not satisfied, and the condition expression is true, then source lines following the /ELSEIF directive are selected to be read. Otherwise, lines are excluded until the next /ELSEIF, /ELSE, or /ENDIF in the same /IF group is encountered.

### **/ELSE (Positions 7-11)**

The /ELSE compiler directive is used to unconditionally select source lines to be read following a failed /IF or /ELSEIF test. The following entries are used for /ELSE:

<b>Positions</b>	<b>Entry</b>
7 - 11	/ELSE
12 - 80	Blank
81 - 100	Comments

If the previous /IF or /ELSEIF was not satisfied, source lines are selected until the next /ENDIF.

If the previous /IF or /ELSEIF was satisfied, source lines are excluded until the next /ENDIF.

### **/ENDIF (Positions 7-12)**

The /ENDIF compiler directive is used to end the most recent /IF, /ELSEIF or /ELSE group. The following entries are used for /ENDIF:

<b>Positions</b>	<b>Entry</b>
7 - 12	/ENDIF
13 - 80	Blank
81 - 100	Comments

Following the /ENDIF directive, if the matching /IF directive was a selected line, lines are unconditionally selected. Otherwise, the entire /IF group was not selected, so lines continue to be not selected.

### **/EOF (Positions 7-10)**

The /EOF compiler directive is used to indicate that the compiler should consider that end-of-file has been reached for the current source file. The following entries are used for /EOF:

<b>Positions</b>	<b>Entry</b>
7 - 10	/EOF
11 - 80	Blank
81 - 100	Comments

/EOF ends any active /IF group that became active during the reading of the current source member. If the /EOF was in a /COPY file, then any conditions that were active when the /COPY directive was read are still active.

**Note:** If excluded lines are being printed on the listing, the source lines continue to be read and listed after /EOF, but the content of the lines is completely ignored by the compiler. No diagnostic messages are ever issued after /EOF.

Using the /EOF directive enhances compile-time performance when an entire /COPY member is to be used only once, but may be copied in multiple times. (This is not true if excluded lines are being printed.)

# /DEFINE and /UNDEFINE

IBM i

/DEFINE condition-name - Adds condition to list of currently defined conditions.

Subsequent /IF DEFINED(condition-name) would be true.

Subsequent /IF NOT DEFINED(condition-name) would be false.

/UNDEFINE condition-name - Removes condition name from list

Subsequent /IF DEFINED(condition-name) would be false.

Subsequent /IF NOT DEFINED(condition-name) would be true.

© Copyright IBM Corporation 2011

Figure 3-5. /DEFINE and /UNDEFINE

AS106.0

## Notes:

The /DEFINE compiler directive defines conditions for conditional compilation. The entries in the condition-name area are free-format (do not have to be left justified).

Positions	Entry
7 - 13	/DEFINE
14	Blank
15 - 80	Condition name
81 - 100	Comments

The /DEFINE directive adds a condition name to the list of currently defined conditions. A subsequent /IF DEFINED(*condition-name*) would be true. A subsequent /IF NOT DEFINED(*condition-name*) would be false.

**Note:** The command parameter **DEFINE** can be used to predefine up to 32 conditions on the **CRTBNDRPG** and **CRTRPGMOD** commands.

You use the /UNDEFINE directive to indicate that a condition is no longer defined. The entries in the condition name area are free-format (do not have to be left justified).

<b>Positions</b>	<b>Entry</b>
7 - 15	/UNDEFINE
16	Blank
17 - 80	Condition name
81 - 100	Comments

The /UNDEFINE directive removes a condition name from the list of currently defined conditions. A subsequent /IF DEFINED(*condition-name*) would be false. A subsequent /IF NOT DEFINED(*condition-name*) would be true.

**Note:** Any conditions specified on the **DEFINE** parameter are considered to be defined when processing /IF and /ELSEIF directives. These conditions can be removed using the /UNDEFINE directive.

/DEFINE and /UNDEFINE example:

### **/DEFINE OL88**

- Adds condition OL88
- Any subsequent /IF DEFINED (OL88) is true
- Any subsequent /IF NOT DEFINED (OL88) is false

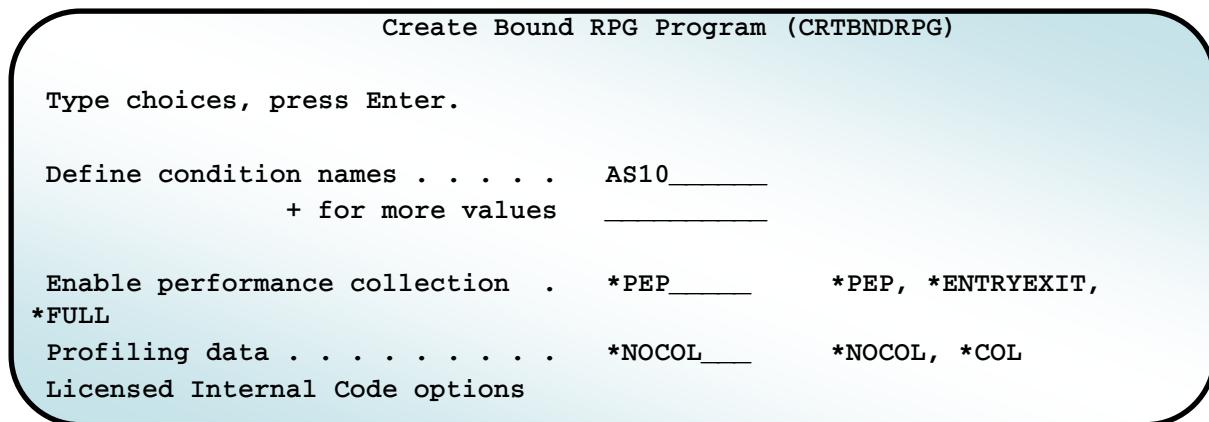
### **/UNDEFINE OL88**

- Removes condition OL88
- Any subsequent /IF DEFINED (OL88) is false
- Any subsequent /IF NOT DEFINED (OL88) is false

# Defining condition names as compilation parameter

IBM i

- Parameter of **CRTBNDPRG** and **CRTRPGMOD**
- Optionally define one or more conditions
- Maximum 32 conditions
- Like /DEFINE compiler directive



© Copyright IBM Corporation 2011

Figure 3-6. Defining condition names as compilation parameter

AS106.0

## Notes:

The **Define condition** parameter specifies condition names that are defined before the compilation begins. Using this parameter is the same as coding the /DEFINE condition-name directive on the first line of the RPG IV source member.

\***NONE**, the default, means no conditions are defined.

Up to 32 condition names can be specified. Each name can be up to 50 characters long. All condition names defined on the compile command are considered to be defined at the start of compilation.

## Rules for testing conditions

IBM i

- /ELSEIF and /ELSE invalid outside /IF group
- /IF group can contain only one /ELSE directive
- /ELSEIF directive cannot follow an /ELSE directive
- /ENDIF invalid outside /IF, /ELSEIF or /ELSE group
- Every /IF must have matching /ENDIF
- All directives associated with /IF group must be in same source file

© Copyright IBM Corporation 2011

Figure 3-7. Rules for testing conditions

AS106.0

### Notes:

It is not valid to have /IF in one copy member and the matching /ENDIF in a second copy member in a nested /COPY. However, a complete /IF group can be in a /COPY nested within another /COPY.

# Predefined conditions

IBM i

```
*ILERPG
  /IF DEFINED(*ILERPG)
  H BNDDIR('QC2LE')
  /ENDIF

*CRTBNDRPG
  / IF DEFINED(*CRTBNDRPG)
  H DFTACTGRP(*NO)
  /ENDIF

*CRTRPGMOD
  /IF DEFINED(*CRTRPGMOD)
  /IF NOT DEFINED(THIS_IS_MAIN)
H NOMAIN
  /ENDIF
  /ENDIF

*VnRnMn
  /IF DEFINED(*V5R1M0)
  // Specify code that is valid in V5R1M0 and subsequent releases
I/INCLUDE SRCFIL,MBR2
  /ELSE
  // Specify code that is available in V4R4M0
I/COPY SRCFIL,MBR2
  /ENDIF
```

© Copyright IBM Corporation 2011

Figure 3-8. Predefined conditions

AS106.0

## Notes:

Predefined conditions can be extremely useful in astuteness where a program is supported in more than one release and requires different compilation options, for example.

## Example: Conditional directives (1 of 2)

IBM i

```

000100 // Declare Files
000200 FItem_PF IF E K Disk
000300 /If defined(*V5R2MO)
000400 FItemInqOV CF E Workstn IndDS(WkstnInd)
000500 /ElseIf defined(*V4R5MO)
000600 FItemInqOVACF E Workstn
000700 /EndIf
000800
000900 /If defined(*V5R2MO)
001000 // Map indicators in DSPF to named indicators
001100 D WkstnInd DS
001200 D NotFound 40 40N
001300 D LowQty 30 30N
001400 D Exit 03 03N
002400 NotFound = Not *found(item_PF); // Set indicator for record not found
002500
002600 If *found(Item_PF); // Item Number valid?
002700   LowQty = (ItmQtyOH + ItmQtyOO) < 20; // Set Indicator for Qty < Minimum
002800   Write Detail; // Write to buffer
002900 Endif;
003000
003100 // Display prompt with error or not
003200 Exfmt Prompt; // Write to buffer; write buffer to display; enable read
003300 Enddo;
003400 // F3 pressed; user wants to exit program
003500 *InLR = *on;
003600 /END-FREE

```

© Copyright IBM Corporation 2011

Figure 3-9. Example: Conditional directives (1 of 2)

AS106.0

### Notes:

Suppose you have users in different locations who are running the same application but are using different levels of IBM i or OS/400. Let's assume that some users are using Version 5 Release (V5R2). Others are using Version 4 Release 5 (V4R5).

What should we do? Do we need to write two separate sets of programs?

No! What we can do is use compiler-provided conditional directives to condition the source. In this visual, you can see that we use a different display file for V5R2 versus V4R5 since named indicators are a V5 feature.

When you compile the program, the value of the target release parameter determines what code is compiled to create the \*PGM object. When you specify \*CURRENT on a V5R2 system, only the code conditioned for the V5R2 compile would be included. The V4R5 code would be excluded.

## Example: Conditional directives (2 of 2)

IBM i

```

003700  /ElseIf defined(*V4R5MO)
003800  C           Eval      PgmNam = 'ITEMINQOV'
003900  C           Write     Header
004000  C           Write     Footer
004100  C           Exfmt    Prompt
004200  C           Dow      NOT *In03
004300  C   ItmNbr   Chain    Item_PF          40
004400
004500  C           If       NOT *In40
004600  ** Display details
004700  C           Eval      *In30 = (ItmQtyOH + ItmQtyOO) < 20
004800  C           Write     Detail
004900  C           EndIf
005000
005100  ** Display prompt with error or not
005200  C           Exfmt    Prompt
005300  C           Enddo
005400
005500  C           Eval      *InLR = *ON
005600  C           Return
005700  /EndIf

```

© Copyright IBM Corporation 2011

Figure 3-10. Example: Conditional directives (2 of 2)

AS106.0

### Notes:

In this visual, you can see the rest of the source member that is conditioned on the compile being done for V4R5.

## Uses of conditional directives

IBM i

- Allow selective copying of prototypes
- Allow same member can hold code to support different compiler releases
- Allow mixed specifications in a single copy member (for example, D-specs, and Calculation specification, for a subroutine)
- Control compilation options (Header specification) dependent on what CRTxxx command is used

© Copyright IBM Corporation 2011

Figure 3-11. Uses of conditional directives

AS106.0

### Notes:

Conditional directives can be used to condition H-spec options depending on what compilation command is used. As you know, prototypes for called procedures, and subprocedures can be held as a group in a single source member. Conditional directives can be used to tell the compiler which specific prototypes to include in the compilation, thus eliminating unreferenced prototypes.

# Machine exercise: Using conditional compiler directives

IBM i



© Copyright IBM Corporation 2011

Figure 3-12. Machine exercise: Using conditional compiler directives

AS106.0

## Notes:

Perform the machine exercise *using conditional compiler directives*.

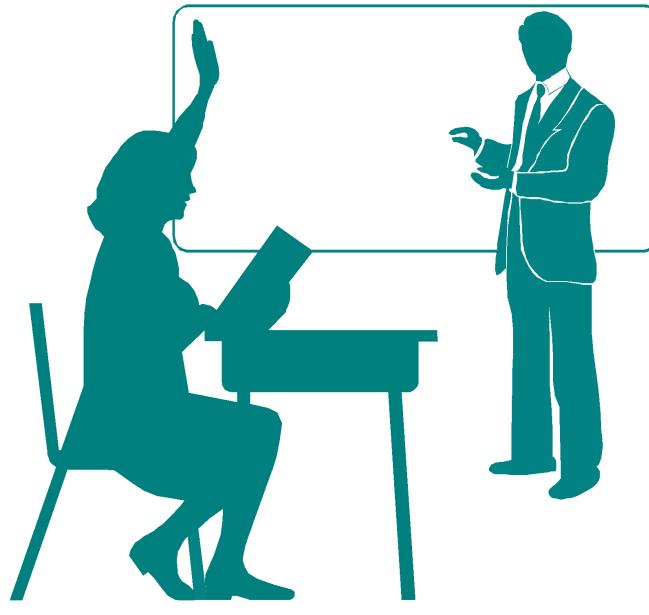


## 4.2. Based variables and pointers

# Introducing based variables

IBM i

- What are *based variables*?
- Why should they be used?
- How can they be used?



© Copyright IBM Corporation 2011

Figure 3-13. Introducing based variables

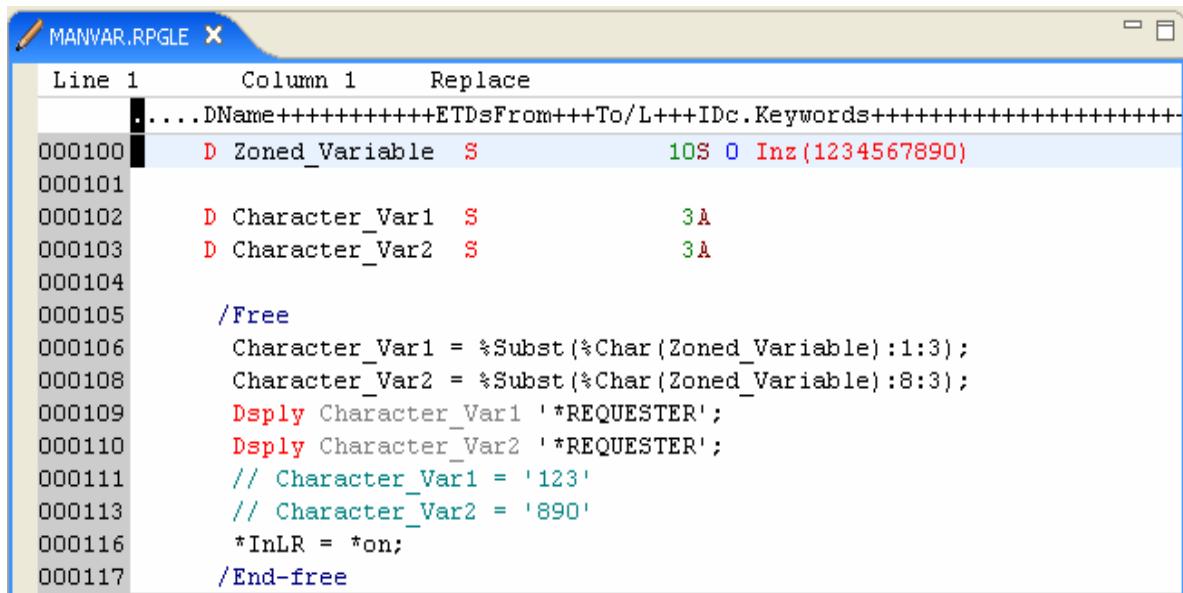
AS106.0

## Notes:

The RPG IV language has many enhanced features to encourage more effective design and coding. *Based variables* and *pointers* present an alternative to using data structures, especially when mapped over storage external to the program (for example, user spaces and heap storage). They also provide an efficient means for decoding data from system APIs and trigger program parameters. This topic gives you something to think about.

# Manipulating variables using EVAL and BIFs

IBM i



```

MANWAR.RPGLE X
Line 1      Column 1      Replace
.....DName++++++ETDsFrom++To/L+++IDc.Keywords+++++
000100    D Zoned_Variable S          10S 0 Inz(1234567890)
000101
000102    D Character_Var1 S          3A
000103    D Character_Var2 S          3A
000104
000105    /Free
000106        Character_Var1 = %Subst(%Char(Zoned_Variable):1:3);
000108        Character_Var2 = %Subst(%Char(Zoned_Variable):8:3);
000109        Dsply Character_Var1 '*REQUESTER';
000110        Dsply Character_Var2 '*REQUESTER';
000111        // Character_Var1 = '123'
000113        // Character_Var2 = '890'
000116        *InLR = *on;
000117    /End-free

```

© Copyright IBM Corporation 2011

Figure 3-14. Manipulating variables using EVAL and BIFs

AS106.0

## Notes:

In this example, we are using EVAL and Built-in functions (BIFs) to convert the contents of a numeric variable to a character variable. Such a conversion may be necessary, for example, to allow us to perform string manipulation on the resulting character values.

The code in the visual:

- Has three separate variables, requiring 16 bytes of working storage
- Uses a relatively expensive EVAL operation to perform the conversion

We can rewrite the above code in a simple, more effective manner.

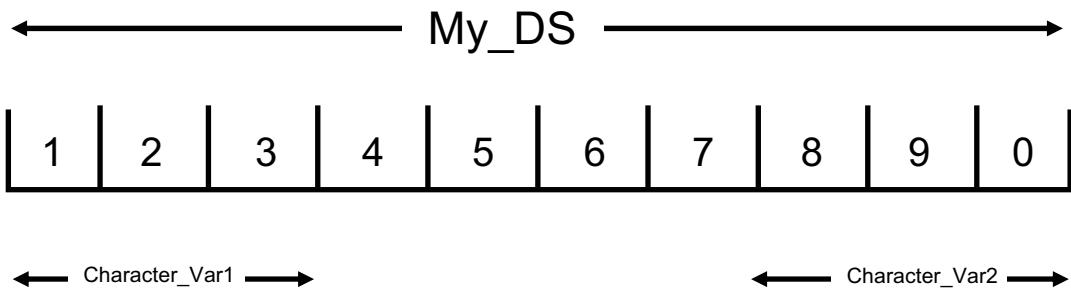
# Manipulating variables using a data structure

IBM i

```

FIG313.RPGLE X
Line 1      Column 1      Replace
.....DName++++++ETDsFrom+++To/L+++IDc.Keywords+++++Comments+++
000100    DMy_DS           DS
000101
000102    D Zoned_Variable          10S 0 Inz(1234567890)
000105
000106    D Character_Var1        3A   Overlay(Zoned_Variable)
000107    D Character_Var2        3A   Overlay(Zoned_Variable : 8)
000109
000110

```



© Copyright IBM Corporation 2011

Figure 3-15. Manipulating variables Using a data structure

AS106.0

## Notes:

Data structures provide a simpler method of achieving the same result.

We have reduced the size of working storage to 10 bytes and no longer require the EVAL and BIF operations.

Through the use of the OVERLAY keyword, we have defined overlapping subfields (**Character\_Var1** and **Character\_Var2**). We therefore have four different windows on the 10 bytes defined:

- Bytes 1-3 and 8-10 of the data structure **MY\_DS**: We could use substring functions to access
- Bytes 1-3 and 8-10 of the major subfield, **Zoned\_Variable**: We could use arithmetic operations to access
- Bytes 1-3 through the subfield **Character\_Var1**
- Bytes 8-10 through the subfield **Character\_Var2**

# Based variables

IBM i

```
D BasedVar      S          3A   BASED (basing_pointer_name)
D basing_pointer_name...
D                  S          *
```

*basing\_pointer\_name* MUST be set to a storage location before BasedVar can be used

© Copyright IBM Corporation 2011

Figure 3-16. Based variables

AS106.0

## Notes:

The data type \* means that this field ***basing\_pointer\_name*** is a pointer.

When the BASED keyword is specified for a data structure or a stand-alone field, a basing pointer is created using the name specified as the keyword parameter. This basing pointer holds the address (storage location) of the based data structure or stand-alone field being defined. In other words, the name specified in positions 7 through 21 is used to refer to the data stored at the location contained in the basing pointer.

**Note:** Before the based data structure or stand-alone field can be used, the basing pointer must be assigned a valid address.

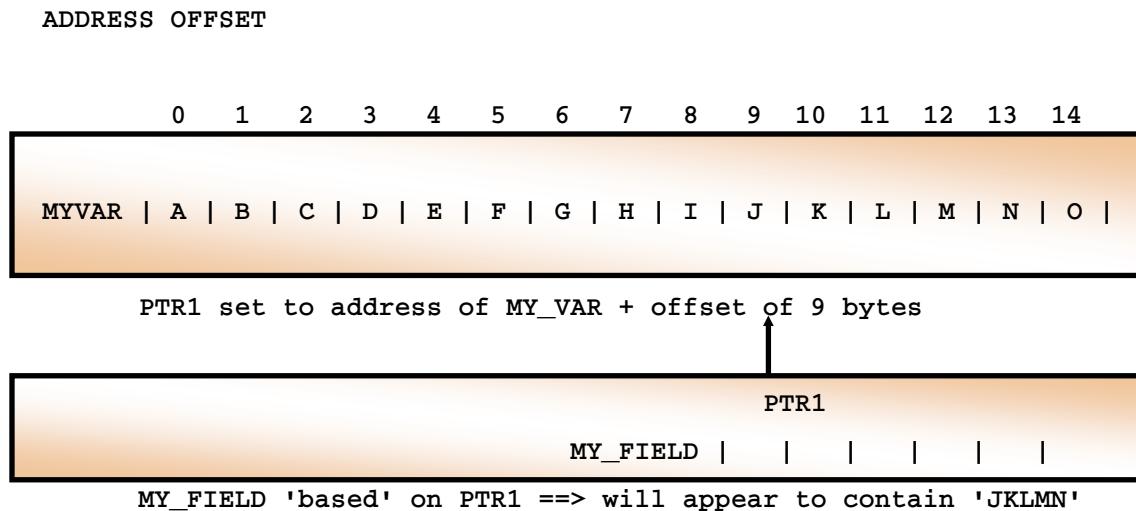
If an array is defined as a based stand-alone field, it must be a run-time array.

If a based field is defined within a subprocedure, then both the field and the basing pointer are local to that subprocedure.

# Basing pointer data type

IBM i

- Locates the storage for based variables.
- Access based variable's data by setting basing pointer variable to point to the required storage location.



© Copyright IBM Corporation 2011

Figure 3-17. Basing pointer data type

AS106.0

## Notes:

Basing pointers are used to locate the storage for based variables. The storage is accessed by defining a field, array, or data structure as based on a particular basing pointer variable and setting the basing pointer variable to point to the required storage location.

For example, consider the based variable *MY\_FIELD*, a character field of length 5, which is based on the pointer *PTR1*. The based variable does not have a fixed location in storage. You must use a pointer to indicate the current location of the storage for the variable.

Use the *BASED* keyword on the definition specification to define a basing pointer for a field. Basing pointers have the same scope as the based field.

The length of the basing pointer field must be 16 bytes long and must be aligned on a 16-byte boundary. This requirement for boundary alignment can cause a pointer subfield of a data structure not to follow the preceding field directly, and can cause multiple occurrence data structures to have non-contiguous occurrences.

The default initialization value for basing pointers is **\*NULL**.

**Note:** When coding basing pointers, you must be sure that you set the pointer to storage that is large enough and of the correct type for the based field.

You can add or subtract an offset from a pointer in an expression, for example,  $PTR=PTR+offset$ . When doing pointer arithmetic, be aware that it is your responsibility to ensure that you are still pointing within the storage of the item you are pointing to. In most cases, no exception is issued if you point before or after the item.

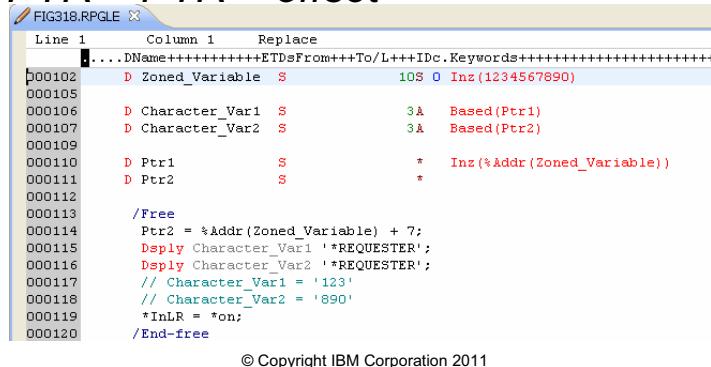
When subtracting two pointers to determine the offset between them, the pointers must be pointing to the same space or the same type of storage. For example, you can subtract two pointers in static storage, two pointers in automatic storage, or two pointers within the same user space.

## Based variable alternative (1 of 2)

IBM i

Set a basing pointer

- Initialize with **INZ(%ADDR(FLD))** where *FLD* is a non-based variable
- Assign the pointer to the result of **%ADDR(X)** where *X* is any variable
- Assign the pointer to the value of another pointer
- Move the pointer forward or backward in storage using pointer arithmetic:  $PTR = PTR + offset$



```

FIG318.RPGLE X
Line 1      Column 1      Replace
.....DName++++++ETDsFrom++To/L++IDc.Keywords+++++
000102     D Zoned_Variable S           10S 0 Inz(1234567890)
000105
000106     D Character_Var1 S           3A   Based(Ptr1)
000107     D Character_Var2 S           3A   Based(Ptr2)
000109
000110     D Ptr1    S           *   Inz(%Addr(Zoned_Variable))
000111     D Ptr2    S           *
000112
000113 /Free
000114     Ptr2 = %Addr(Zoned_Variable) + 7;
000115     Disp Character_Var1 '*REQUESTER';
000116     Disp Character_Var2 '*REQUESTER';
000117     // Character_Var1 = '123'
000118     // Character_Var2 = '890'
000119     *InLR = *on;
000120 /End-free

```

© Copyright IBM Corporation 2011

Figure 3-18. Based variable alternative (1 of 2)

AS106.0

### Notes:

*Offset*, ( $PTR = PTR + offset$ ), is the distance in bytes that the pointer is moved.

This example produces the same results as those shown earlier. However, we are using pointer variables *PTR1* and *PTR2* to base *Character\_Var1* and *Character\_Var2* upon the variable **Zoned\_Variable**.

This is the same approach as that taken with the data structure.

*Character\_Var1* and *Character\_Var2* overlay *Zoned\_Variable*.

Note that the variables *PTR1* and *PTR2* have no declared length. Pointer variables are always 16 bytes in length. These variables hold the address of a storage location, not the value in the location, the normal behavior of a variable.

In order to manipulate the based variable (for example, *Character\_Var1*), the pointer (*Ptr1*) must first be resolved; it must point to a valid address. We have achieved this by using the **%ADDR** built-in function:

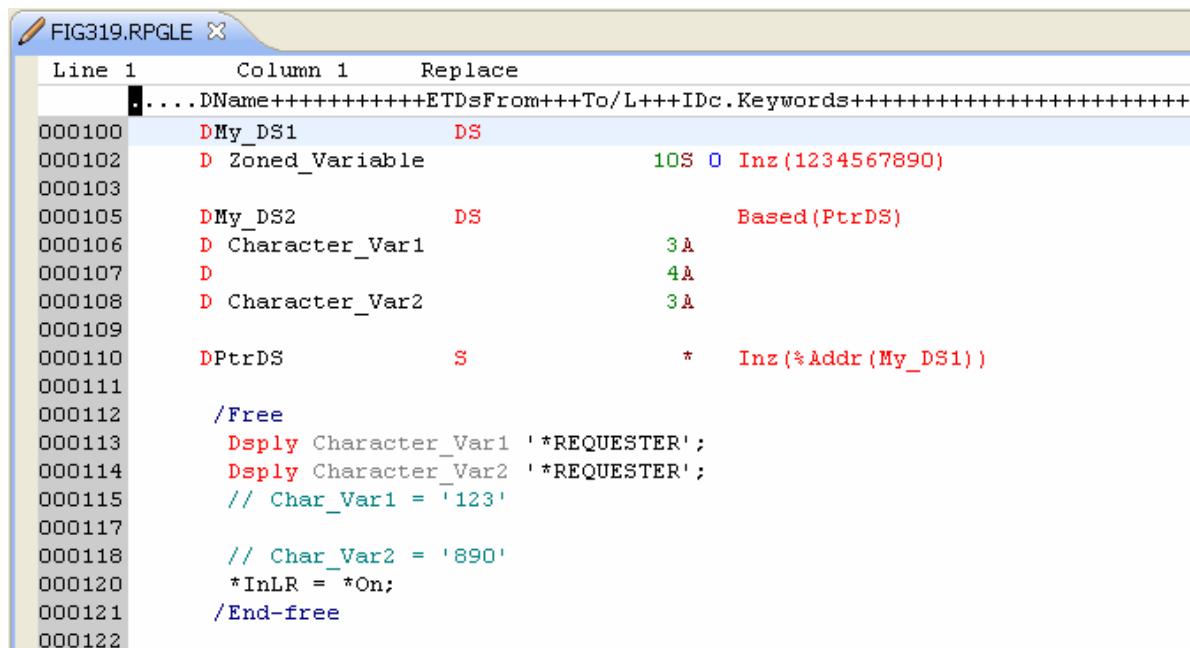
- *PTR1*: On the D-spec (Initialize (INZ) keyword)

- *PtTR2*: On a C-spec EVAL operation

Note that we can manipulate the address held within a pointer variable by adding an offset value. Thus *Character\_Var2* can be made to overlay the last three bytes of *Zoned\_Variable*. More than one variable can be based on a single pointer if desired.

## Based variable alternative (2 of 2)

IBM i



```

FIG319.RPGLE X
Line 1      Column 1      Replace
. .... DName++++++ETDsFrom+++To/L+++IDc.Keywords+++++++
000100  DMy_DS1           DS
000102  D Zoned_Variable   10S 0 Inz(1234567890)
000103
000105  DMy_DS2           DS          Based(PtrDS)
000106  D Character_Vari    3A
000107  D                   4A
000108  D Character_Var2    3A
000109
000110  DPtrDS             S          * Inz(%Addr(My_DS1))
000111
000112  /Free
000113  Dspla Character_Vari '*REQUESTER';
000114  Dspla Character_Var2 '*REQUESTER';
000115  // Char_Vari = '123'
000117
000118  // Char_Var2 = '890'
000120  *InLR = *On;
000121  /End-free
000122

```

© Copyright IBM Corporation 2011

Figure 3-19. Based variable alternative (2 of 2)

AS106.0

### Notes:

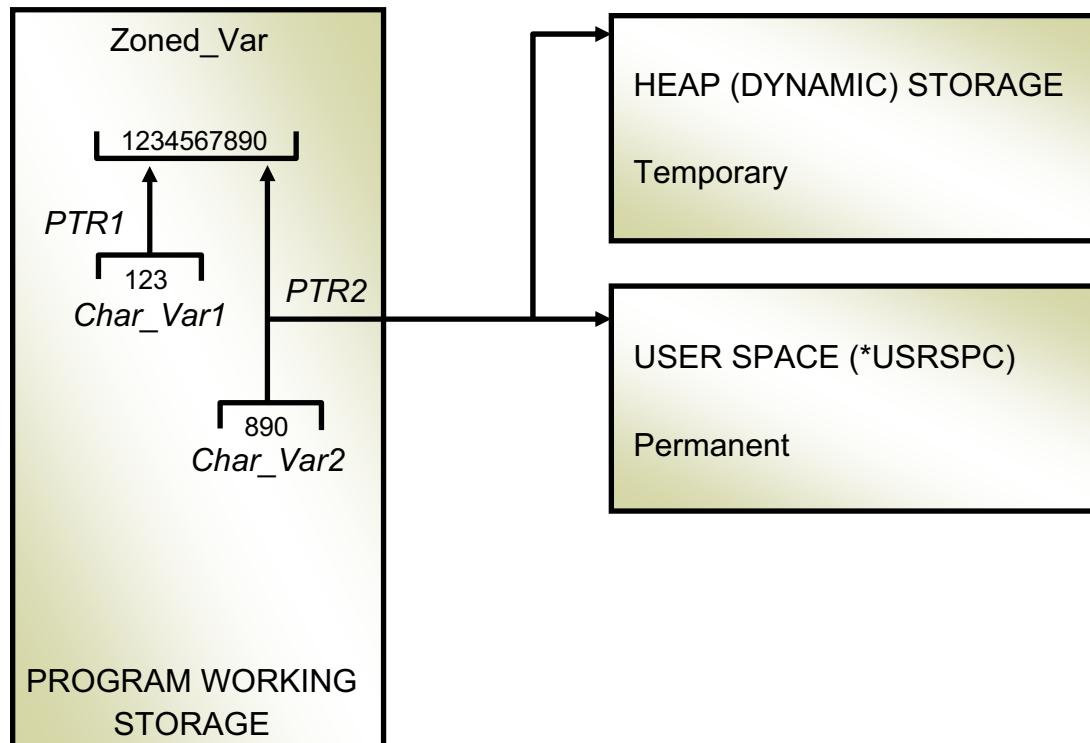
An entire data structure can be defined as based. This provides you with the capability to overlay one data structure upon another, an operation not otherwise possible.

To achieve the same result without based variables, you would have to use the EVAL and BIFs opcodes to copy data from one data structure to another.

## 4.3. Dynamic storage allocation and pointers

# Pointing to heap storage and user spaces

IBM i



© Copyright IBM Corporation 2011

Figure 3-20. Pointing to heap storage and user spaces

AS106.0

## Notes:

Our use of pointers has been limited so far to overlaying variables in the space reserved for the program's working storage:

- Program static storage area (PSSA)
- Program automatic storage area (PASA)

Pointers can be resolved to areas outside the PSSA/PASA:

- System heap (dynamic) storage
- User spaces

In other words, storage can be allocated outside the program as required.

## Managing dynamically allocated storage

ILE allows you to directly manage run-time storage from your program by managing heaps. A *heap* is an area of storage used for allocations of dynamic storage. The amount of dynamic storage required by an application depends on the data being processed by the programs and procedures that use the heap.

You manage heaps by using the storage management operations **ALLOC** (or **%Alloc**), **REALLOC**, and **DEALLOC**, or by using ILE bindable APIs.

You are not required to explicitly manage run-time storage. However, you may want to do so if you want to make use of dynamically allocated run-time storage. For example, you may want to do this if you do not know exactly how large an array or multiple-occurrence data structure should be. You could define the array or data structure as **BASED** and acquire the actual storage for the array or data structure once your program determines how large it should be.

There are two types of heaps available on the system: a *default heap* and a *user-created heap*. The RPG storage management operations use the default heap. The following sections show how to use RPG storage management operations with the default heap and also how to create and use your own heap using the storage management APIs. For more information on user-created heaps and other ILE storage management concepts, refer to *ILE Concepts*.

# Heap request API example: CEEGTST API parameters

IBM i

Name	Use	Type
Heap identifier	Input	Binary(4)
Size	Input	Binary(4)
Address pointer	Output	*
Feedback code (optional)	Output	Char(12)

© Copyright IBM Corporation 2011

Figure 3-21. Heap request API example: CEEGTST API parameters

AS106.0

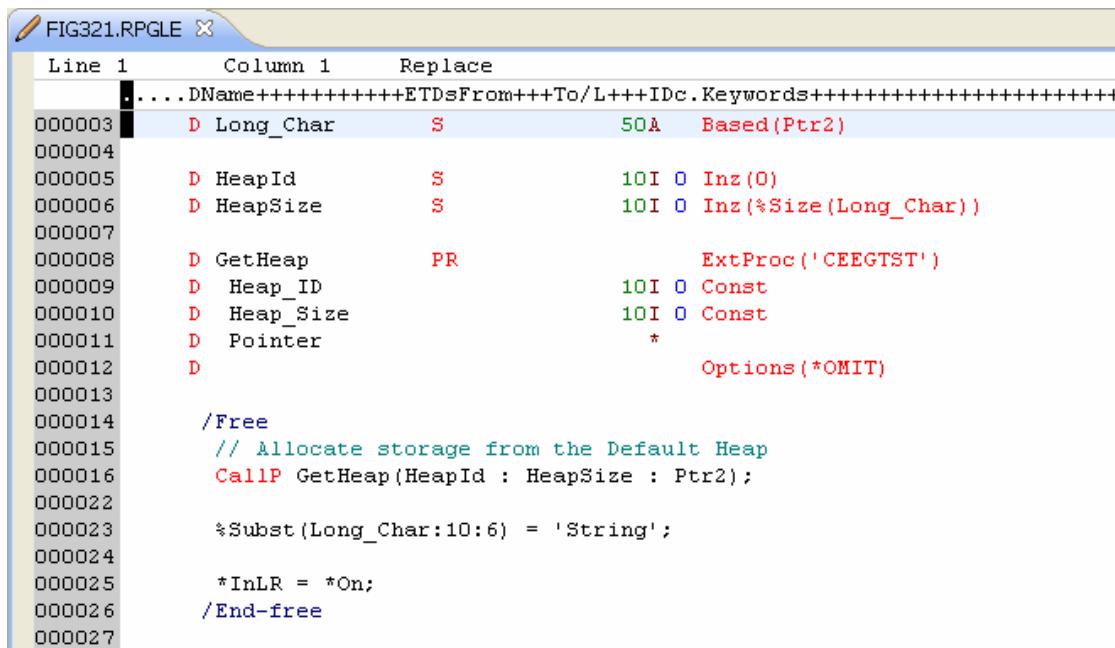
## Notes:

The Get Heap Storage (CEEGTST) API allocates storage within a heap. Three parameters are required in order for this API to allocate heap storage, and one, the feedback code, is optional.

RPG IV allows the I and U data types to be used for binary. A **Binary(4)** value can be defined as a 10-byte integer.

# Using the heap: System APIs

IBM i



```

FIG321.RPGLE X
Line 1      Column 1      Replace
.....DName++++++ETDsFrom+++To/L+++IDc.Keywords+++++
000003     D Long_Char      S          50A   Based(Ptr2)
000004
000005     D HeapId         S          10I 0 Inz(0)
000006     D HeapSize        S          10I 0 Inz(%Size(Long_Char))
000007
000008     D GetHeap          PR          ExtProc('CEEGTST')
000009     D Heap_ID          Const       10I 0 Const
000010     D Heap_Size         Const       10I 0 Const
000011     D Pointer           *          *
000012             Options(* OMIT)
000013
000014 /Free
000015 // Allocate storage from the Default Heap
000016 CallP GetHeap(HeapId : HeapSize : Ptr2);
000022
000023 *Subst(Long_Char:10:6) = 'String';
000024
000025 *InLR = *On;
000026 /End-free
000027

```

© Copyright IBM Corporation 2011

Figure 3-22. Using the heap: System APIs

AS106.0

## Notes:

We can use the CEEGTST API to request an allocation of storage from the system heap.

We have asked for 50 bytes to be allocated from the default heap (HeapId = 0). We could have allocated storage from a specific heap, but this requires more programming effort. Once the storage is allocated, we can assign and manipulate values using the based variable as normal.

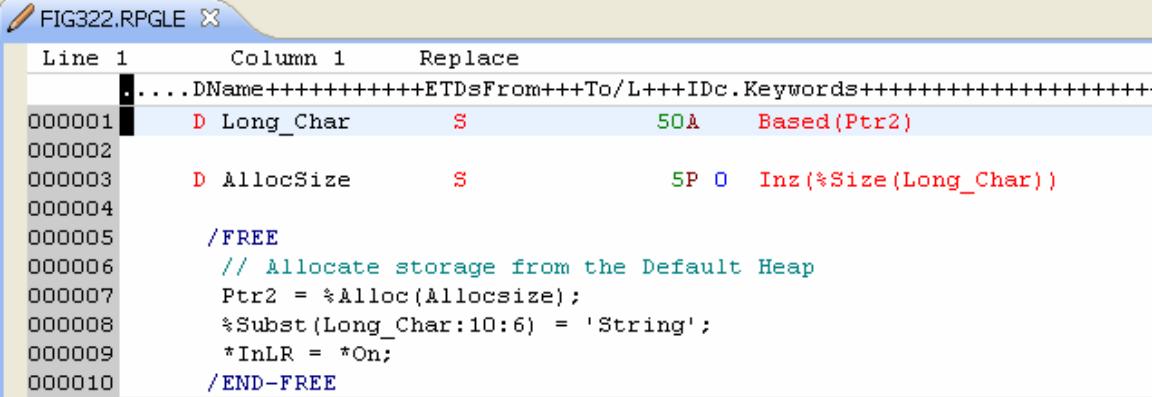
One of the parameters we supply to the API is a pointer variable, (*PTR2*). It returns the address of the allocated storage. Note that *PTR2* does not have to be explicitly declared on the D-specs.

As with most APIs, numeric parameters are defined as binary integers. Note that the API is one of the CEExxxx family. These are ILE language (bindable) APIs, requiring a procedure call (bound call) rather than a program call (dynamic call).

In order to use a bound CALLP (shown above), the program must be created to run in a non-default activation group (ILE).

# Using the heap: RPG IV operations

IBM i



```

Line 1      Column 1      Replace
.....DName++++++ETDsFrom+++To/L+++IDc.Keywords+++++
000001      D Long_Char      S          50A      Based(Ptr2)
000002
000003      D AllocSize      S          5P 0    Inz(%Size(Long_Char))
000004
000005      /FREE
000006      // Allocate storage from the Default Heap
000007      Ptr2 = %Alloc(Allocsize);
000008      *Subst(Long_Char:10:6) = 'String';
000009      *InLR = *On;
000010      /END-FREE

```

© Copyright IBM Corporation 2011

Figure 3-23. Using the heap: RPG IV operations

AS106.0

## Notes:

The ALLOC (we show the equivalent %Alloc BIF), REALLOC, and DEALLOC opcodes can be used instead of the ILE APIs, making it much simpler to allocate storage from the system heap.

The %Alloc BIF or ALLOC opcode can be used to allocate storage from the default heap only, though this is unlikely to be a limitation in practice. Use of this opcode avoids the need for a procedure call (for example, CALLB) to the ILE APIs and means the program can be created to run in the default activation group.

### ALLOC opcode or %Alloc BIF

The ALLOC operation or %Alloc BIF allocates storage in the default heap of the length specified in factor 2. The result field pointer is set to point to the new heap storage. The storage is not initialized.

Factor 2 must be a numeric with zero decimal positions. It can be a literal, constant, stand-alone, field, subfield, table name, or array element. The value must be between 1 and 16776704. If the value is out of range at run time, an error occurs with status 425. If the

storage could not be allocated, an error occurs with status 426. If these errors occur, the result field pointer remains unchanged.

The result must be a basing pointer scalar variable (a stand-alone field, data structure subfield, table name, or array element).

To handle exceptions with program status codes 425 or 426, the operation code extender E can be specified.

### **REALLOC (Reallocate Storage with New Length) or %Realloc BIF**

The REALLOC operation or %Realloc BIF changes the length of the heap storage pointed to by the result-field pointer to the length specified in factor 2. The result field of REALLOC contains a basing pointer variable. The result field pointer must contain the value previously set by a heap-storage allocation operation (either an ALLOC or REALLOC operation in RPG or some other heap-storage function such as CEEGTST). It is not sufficient to simply point to heap storage; the pointer must be set to the beginning of an allocation.

New storage is allocated of the specified size, and the value of the old storage is copied to the new storage. Then the old storage is deallocated. If the new length is shorter, the value is truncated on the right. If the new length is longer, the new storage to the right of the copied data is uninitialized.

The result field pointer is set to point to the new storage.

If the operation does not succeed, an error condition occurs, but the result field pointer will not be changed. If the original pointer was valid and the operation failed because there was insufficient new storage available (status 425), the original storage is not deallocated, so the result field pointer is still valid with its original value.

If the pointer is valid but it does not point to storage that can be deallocated, then status 426 (error in storage management operation) will be set.

To handle exceptions with program status codes 425 or 426, either the operation code extender E or an error indicator ER can be specified, but not both.

Factor 2 contains a numeric variable or constant that indicates the new size of the storage (in bytes) to be allocated. Factor 2 must be numeric with zero decimal positions. The value must be between 1 and 16776704.

### **DEALLOC (Free Storage)**

The DEALLOC operation frees one previous allocation of heap storage. The result field of DEALLOC is a pointer that must contain the value previously set by a heap-storage allocation operation (either an ALLOC operation in RPG or some other heap-storage allocation mechanism). It is not sufficient to simply point to heap storage; the pointer must be set to the beginning of an allocation.

The storage pointed to by the pointer is freed for subsequent allocation by this program or any other in the activation group.

If operation code extender *N* is specified, the pointer is set to **\*NULL** after a successful deallocation.

To handle DEALLOC exceptions (program status code 426), either the operation code extender *E* or an error indicator *ER* can be specified, but not both. The result field pointer will not be changed if an error occurs, even if *N* is specified.

The result field must be a basing pointer scalar variable (a stand-alone field, data structure subfield, table name or array element).

No error is given at runtime if the pointer is already **\*NULL**.

## User space highlights

IBM i

- Permanent IBM i object (\*USRSPC)
- Like a large \*DTAARA: Up to 16 MB
- Fast to process: Used by List APIs
- Can auto-extend
- Can be saved/restored
- Manipulated by APIs: No commands

Can be accessed using a pointer

© Copyright IBM Corporation 2011

Figure 3-24. User space highlights

AS106.0

### Notes:

Storage allocated from the system heap is temporary only, similar to the local data area (\*LDA). Also like the \*LDA, such storage is only accessible by programs in the same job.

User spaces, on the other hand, are permanent objects, similar to a user data area. These can be very large and can be extended up to a maximum of 16 MB. Compare this size to a character User Data Area of 2,000 bytes.

Apart from the differences in size, a user space is distinguished from a user data area by the fact that the user space is manipulated solely by system APIs. The only IBM i command that supports user spaces is **DLTUSRSPC**.

## User space APIs

IBM i

QUSCRTUS	Create a user space
QUSDLTUS	Delete a user space
QUSRTVUS	Retrieve data from a user space
QUSCHGUS	Change the data in a user space
QUSRUSAT	Retrieve user space attributes
QUSCUSAT	Change user space attributes
QUSPTRUS	Retrieve a pointer to a user space

© Copyright IBM Corporation 2011

Figure 3-25. User space APIs

AS106.0

### Notes:

Before you can manipulate a user space, you must first create it.

# Creating a user space: QUSCRTUS

IBM i

```

000003      D Message      C          'Unable to create User Space'
000005
000006      D SpaceName     S          20A  Inz('MYSPACE    *CURLIB')
000007      D Attribute     S          10A  Inz('A_SPACE')
000008      D Size          S          10I 0  Inz(50)
000009      D InitValue     S          1A   Inz('*')
000010      D Authority     S          10A  Inz('*USE')
000011      D Text          S          50A  Inz('My User Space')
000012      D Replace        S          10A  Inz('*YES')
000013      D ErrorCode      DS         10I 0  Inz(%Size(ErrorCode))
000014      D BytesAvl      PR         20A  Const
000015      D BytesRet      PR         10I 0
000016      D MsgId         7A
000017      D Reserved       1A
000018      D MsgDta        84A
000019
000020      D CreateUSpace   PR         Extpgm('QUSCRTUS')
000021      D
000022      D
000023      D
000024      D
000025      D
000026      D
000027      D
000028      D
000029
000030      /Free
000031      CallP CreateUSpace(SpaceName:Attribute:Size:InitValue:Authority:
000032                      Text:Replace:Errorcode);
000033
000034      If BytesRet <> 0;
000035          Dsplay Message '*EXT';
000036      Endif;
000037      *InLR = *On;
000038
000039      /End-free
000040
000041
000042
000043
000044
000045
000046
000047

```

© Copyright IBM Corporation 2011

Figure 3-26. Creating a user space: QUSCRTUS

AS106.0

## Notes:

This code segment creates the user space MYSPACE using the QUSCRTUS API. The last two parameters for this API are optional but are useful to provide simple error-checking facilities.

We have created a user space 50 bytes in length and filled it with asterisk (\*) characters.

We now wish to manipulate the contents, using it to hold information which we can access in the program. We could achieve this in any programming language using the APIs. With the benefit of pointers and based variables in RPG IV, we can use the QUSPTRUS API to access the contents directly.

# Accessing a user space: QUSPTRUS

IBM i

```

000003      D Message2      C           'Unable to access User Space'
000004
000005      D Long_Char     S           50A   Based(Ptr2)
000006      D Ptr2          S           *                 *
000007      D Short_Char   S           1A    Based(Ptr3)
000008      D Ptr3          S           *                 *
000009
000010      D SpaceName     S           20A   Inz('MYSPACE      *CURLIB')
000011      D ErrorCode      DS
000012      D BytesAvl      10I 0  Inz(*Size(ErrorCode))
000013      D BytesRet      10I 0
000014      D MsgId         7A
000015      D Reserved       1A
000016      D MsgDta        84A
000017
000018      D AccessUSpace  PR           ExtPgm('QUSPTRUS')
000019      D SpaceName     20A   Const
000020      D Ptr2          *           Const
000021      D ErrorCode      100   Const Options(*Nopass)
000022
000023      /Free
000024      CallP AccessUSpace (Spacename : Ptr2 : ErrorCode);
000028
000030      If BytesRet <> 0;
000032          Dspla Message2 '*EXT';
000033          *InLR = *ON;
000034          Return;

```

© Copyright IBM Corporation 2011

Figure 3-27. Accessing a user space: QUSPTRUS

AS106.0

## Notes:

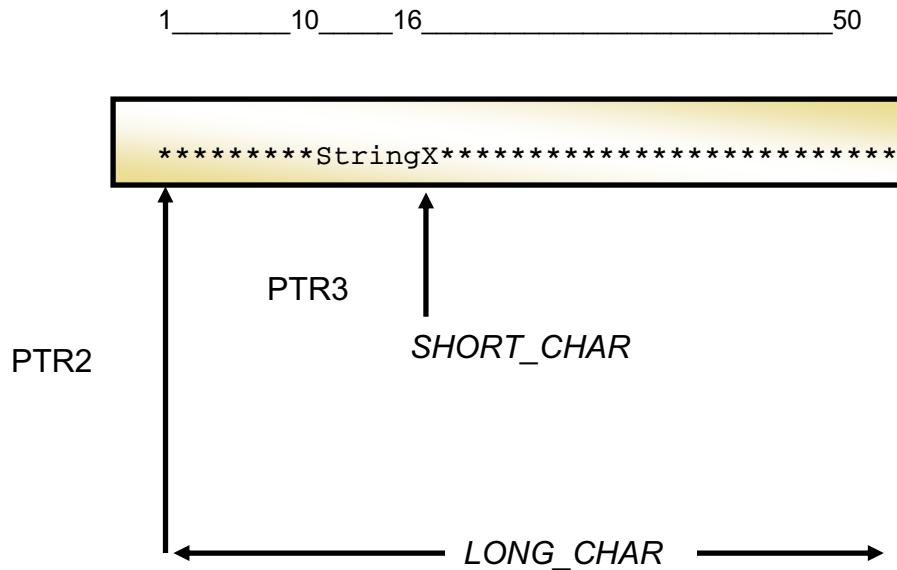
Using **QUSPTRUS**, we have resolved the Pointer PTR2 to the user space object (MYSPACE).

Subsequent manipulation of any variables based on PTR2 results in direct changes to the user space data. There is no need to retrieve or change the data. The use of based variables therefore provides a much more efficient means of processing a user space than the conventional read/write approach.

As a result of the above code, the contents of the user space now appear, as shown on the next visual.

# Contents of My\_Space

IBM i



© Copyright IBM Corporation 2011

Figure 3-28. Contents of My\_Space

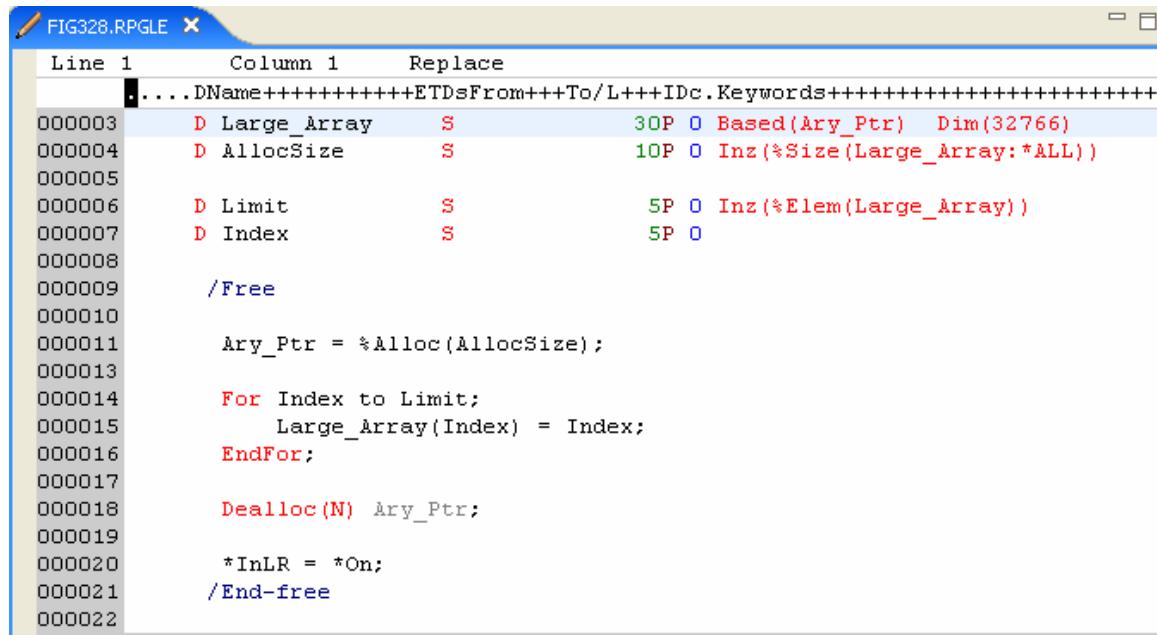
AS106.0

## Notes:

The user space contains *StringX* in positions 10-16 as a result of two simple assignments to base variables *LONG\_CHAR* and *SHORT\_CHAR*.

# Using based variables: Big arrays

IBM i



```

FIG328.RPGLE X
Line 1      Column 1      Replace
.....DName++++++ETDsFrom+++To/L+++IDc.Keywords+++++++
000003      D Large_Array     S          30P 0 Based(Ary_Ptr) Dim(32766)
000004      D AllocSize       S          10P 0 Inz(%Size(Large_Array:*ALL))
000005
000006      D Limit           S          5P 0 Inz(%Elem(Large_Array))
000007      D Index           S          5P 0
000008
000009      /Free
000010
000011      Ary_Ptr = %Alloc(AllocSize);
000013
000014      For Index to Limit;
000015          Large_Array(Index) = Index;
000016      EndFor;
000017
000018      Dealloc(N) Ary_Ptr;
000019
000020      *InLR = *On;
000021      /End-free
000022

```

© Copyright IBM Corporation 2011

Figure 3-29. Using based variables: Big arrays

AS106.0

## Notes:

We have allocated the space for the array when required, and deallocated when no longer needed in the program. The pointer has been set to null once the storage has been released back to the system heap.

## Problems with pointers

IBM i

- Pointer unresolved: \*NULL
- Falling off the edge

Do not point to places you should not!

© Copyright IBM Corporation 2011

Figure 3-30. Problems with pointers

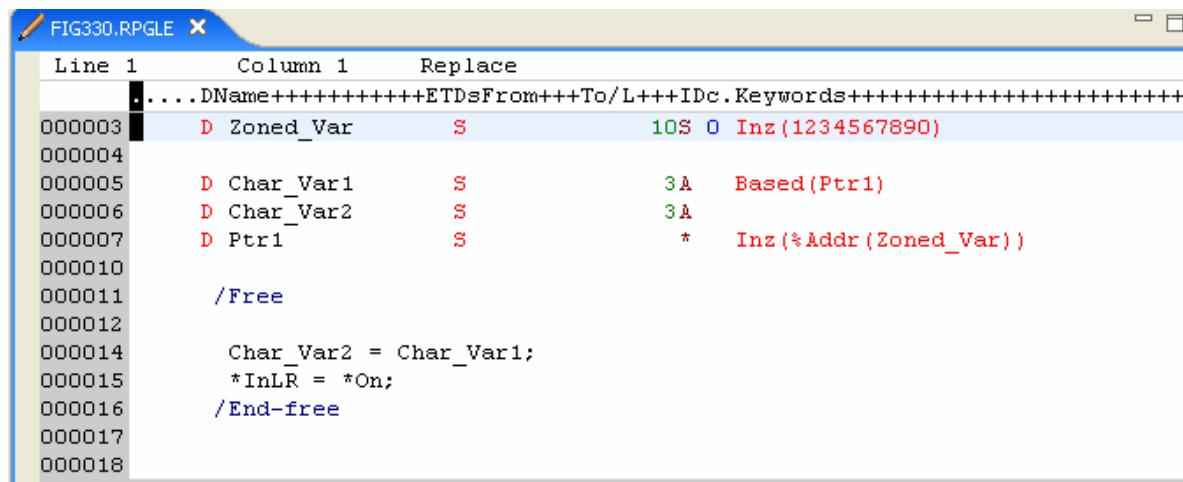
AS106.0

### Notes:

Pointers can present problems to you if you do not use them correctly. Unexpected and unpredictable results can result from pointing to the wrong address.

# Unresolved pointer

IBM i



The screenshot shows an IBM i RPGLE editor window titled 'FIG330.RPGLE'. The code is as follows:

```

Line 1      Column 1      Replace
.....DName++++++ETDsFrom+++To/L+++IDc.Keywords+++++
000003      D Zoned_Var      S          10S 0 Inz(1234567890)
000004
000005      D Char_Var1     S          3A   Based(Ptr1)
000006      D Char_Var2     S          3A
000007      D Ptr1          S          *    Inz(%Addr(Zoned_Var))
000010
000011      /Free
000012
000014      Char_Var2 = Char_Var1;
000015      *InLR = *On;
000016      /End-free
000017
000018

```

```

Message ID . . . . . : RNQ0222      Severity . . . . . : 99
Message . . . . : Pointer or parameter error (C G D F).

```

© Copyright IBM Corporation 2011

Figure 3-31. Unresolved pointer

AS106.0

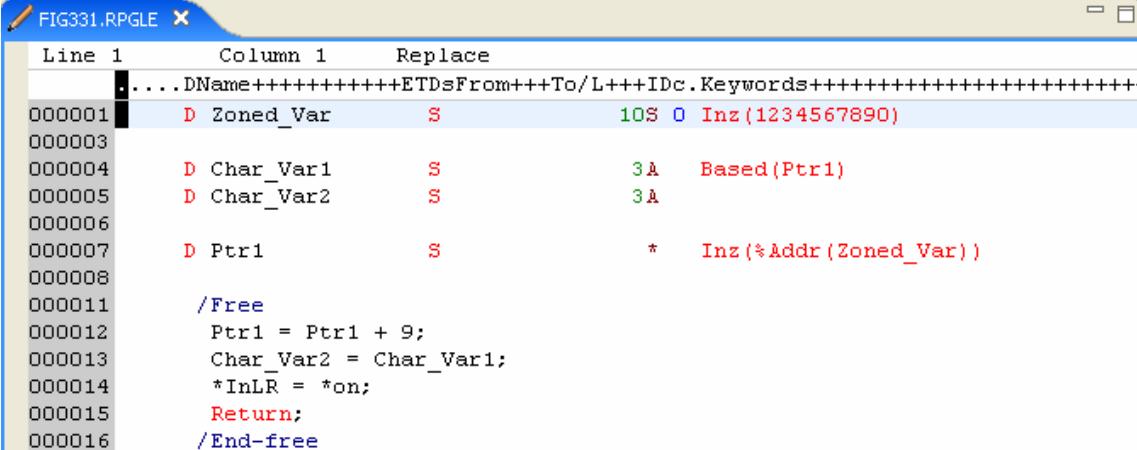
## Notes:

A pointer must be resolved (point to something) before any variable based upon it can be used in an expression.

Pointers are initialized to **\*NULL** by default.

# Falling off the edge

IBM i



```

FIG331.RPGLE X
Line 1      Column 1      Replace
.....DName++++++ETDsFrom++To/L+++IDc.Keywords+++++
000001     D Zoned_Var      S          10S 0 Inz(1234567890)
000003
000004     D Char_Var1      S          3A   Based(Ptr1)
000005     D Char_Var2      S          3A
000006
000007     D Ptr1           S          *   Inz(%Addr(Zoned_Var))
000008
000011     /Free
000012     Ptr1 = Ptr1 + 9;
000013     Char_Var2 = Char_Var1;
000014     *InLR = *on;
000015     Return;
000016     /End-free

```

NAME	ATTRIBUTES	VALUE
CHAR_VAR1	CHAR (3)	'0A'
CHAR_VAR2	CHAR (3)	'0A'
PTR1	POINTER	SPP:E37D39D72B000469
ZONED_VAR	ZONED (10,0)	1234567890.

© Copyright IBM Corporation 2011

Figure 3-32. Falling off the edge

AS106.0

## Notes:

We have some erroneous results in *CHAR\_VAR1* as a result of basing the variable beyond the bounds of variable *ZONED\_VAR*.

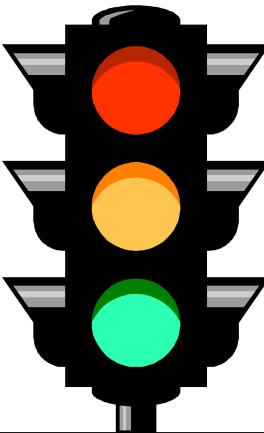
At best, we might obtain a pointer violation condition if we were based on a user space, for example. At worst, we could unknowingly corrupt the information held in variables in surrounding storage.

Thus, we say we have *fallen off the edge*.

## Recommendations

IBM i

- RPG IV based variables and pointers provide:
  - Effective use of storage
  - Simple and efficient programming



Use with caution

© Copyright IBM Corporation 2011

Figure 3-33. Recommendations

AS106.0

### Notes:

Pointers can present problems to the programmer when not used correctly. Unexpected and unpredictable results can result from pointing to the wrong address.

## 4.4. User spaces and list APIs

## List APIs

IBM i

- Like **DSPxxx OUTPUT(\*OUTFILE)** commands
- L in fourth position of API name
- Some list APIs are:
  - QUSLMBR List database file members
  - QDBLDBR List database relations
  - QUSLFLD List fields
  - QBNLPGMI List ILE program information
  - QUSLJOB List job
  - QMHLJOBL List job log messages
  - QWCLOBJL List object locks
  - QUSLOBJ List objects

© Copyright IBM Corporation 2011

Figure 3-34. List APIs

AS106.0

### Notes:

List APIs are similar to the DSPxxx commands that output to a file. List APIs have an **L** in position 4 of their names.

# Using list APIs

IBM i

Steps to follow:

- Create a user space
- Fill user space with output of list API
- Retrieve control information
- Retrieve list entries from user space

© Copyright IBM Corporation 2011

Figure 3-35. Using list APIs

AS106.0

## Notes:

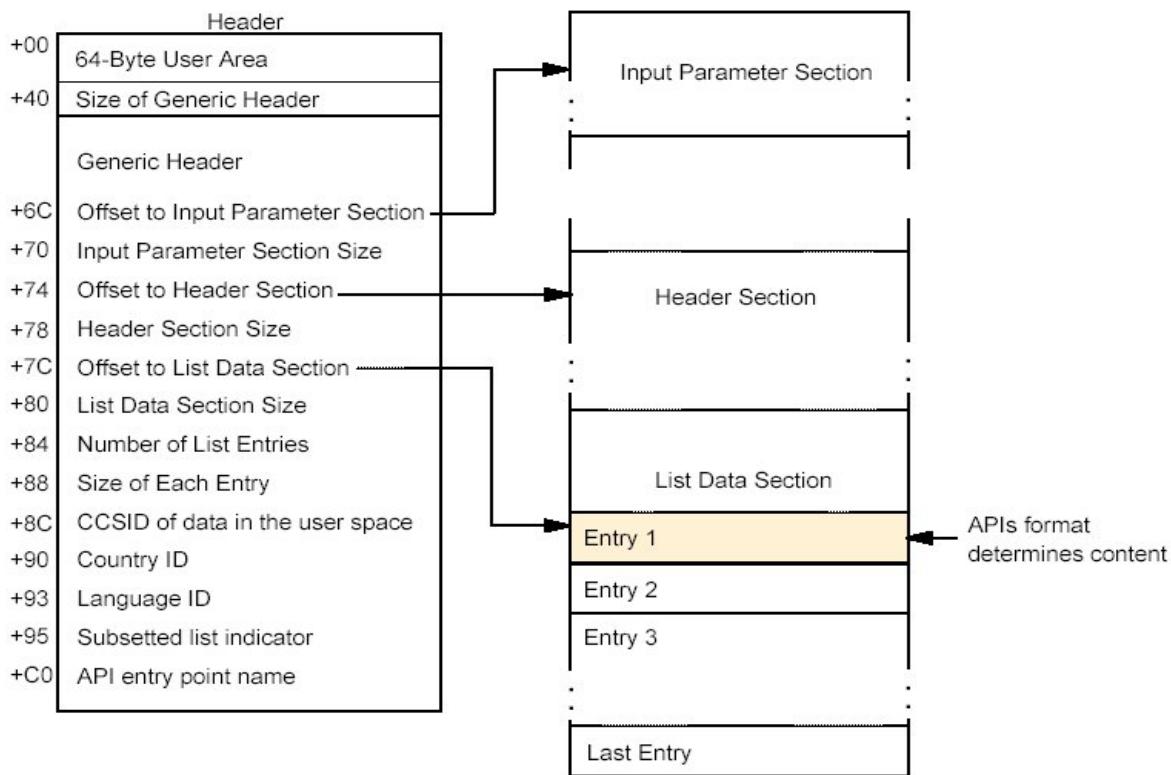
List APIs can retrieve a great deal of information. Hence, user spaces are the ideal place to store their output.

The control information is contained within the generic header block of the user space. This header is the key to unlocking the list data items. Information on offsets, number of entries, and entry length is provided — and this gives the list API its flexibility across version/release updates of IBM i.

In order to use a list API, you must have created a user space that holds its output. Also, you must use the API calls that we discussed earlier to create the user space and to retrieve the contents of the user space.

# Output of list API to user space

IBM i



© Copyright IBM Corporation 2011

Figure 3-36. Output of list API to user space

AS106.0

## Notes:

All list APIs format the data in a user space using a very strict structure.

The data comprises four elements:

- **Generic header**: Contains the necessary control information with which to decode the other three sections
- **Input parameter section** (not shown)
- **List-specific header section** (not shown)
- **List data section**: Contains the data entries

Use **Offset to list data** to determine the position of the first entry in the user space.

**Number of list entries** enables each entry to be processed in a loop, while **Size of each entry** allows the offset to the next entry to be calculated.

This information can be extracted from the generic header portion of the user space data using the QUSRTVUS API. This same API can then be used repeatedly to extract individual entries from the **List data section**.

## Additional Information

The information below is included for your reference.

### Characteristics of a list API

List APIs return information to a user space. List APIs generally have a user space parameter that uses a general (or common) data structure. You must use the general data structure to get at the information placed in the user space by the list API.

#### General data structure

All list APIs have an input parameter section, a header section, and a list data section.

#### User area

The first field in the general data structure is called the *user area*. This is a 64-byte field that is not used or changed by the system. Whatever information you place in this field remains there. For example, you may be able to specify the date last used, include comments about the list, and so forth.

#### Size of generic header

The size of the generic header does not include the size of the user area. All sections have a size, which may differ for each API.

Some fields may be added to the generic header from release to release. Because fields may be added, you may want to check the size of this field. If your application works across multiple releases, it is recommended that you check the size of this field to determine which fields are applicable.

#### Offset to input parameter section

The offset to input parameter section is an offset to the start of the input parameter section. The input parameter section often contains a copy of the input parameters that you pass to the list API. This is a good place to check when things go wrong. You can check what was actually passed to the API.

The input parameter section contains a copy of the continuation handle value that you passed as the continuation handle parameter to the API.

#### Offset to header section

The header section includes an offset to where the header section starts and the size of the header section. This section is needed in the event any input parameters have a special value. The fields in the header section tell what the special value resolved to.

This section is also sometimes used for API-specific control information that is not related to a particular fist entry.

#### Offset to list data section

The offset to the list data section is the offset to the start of the format. The specific format that the API uses is determined by the name you specify for the format name parameter. The specific format that you use determines what information is returned in the user space.

The number of list entries field tells how many entries have been returned to you. The size of each entry field within the list data section tells how large each entry is. In the list data section, each entry is of the same length for a given list. If the size of each entry field is 0, the entries have different lengths, and the format tells the length of each entry.

### Other fields of the generic header

The field called **structure's release and level** is part of the generic header. This field tells the layout of the generic header. For an original program model (OPM) layout, this value should be 0100. For an Integrated Language Environment (ILE) model layout, the value should be 0300.

The information status field tells you whether the information in the user space is complete and accurate, or partial. You need to check the value of this field before you do anything with the information in the user space. Possible values for this field follow:

**C:** Complete and accurate.

**I:** Incomplete. The information you received is not accurate or complete.

**P:** Partial but accurate. The information you received is accurate, but the API had more information to return than the user space could hold.

If you receive the value **P**, you need to process the current information in the user space before you get the remaining information. The API returns a continuation handle, usually in the form of a parameter. You can use this continuation handle value to have the remaining information placed in the user space. You specify the continuation handle value that the API returned as the value of the continuation handle input parameter on your next call to the API.

If the API does not have a continuation handle and the information status field value is **P**, you must further qualify what you want in the list. In other words, you must be more specific on the parameter values that you pass to the API.

Some APIs, such as QUSLFLD, provide an additional API-specific header section. This API is equivalent to **DSPFFD**. The list section gives details of each field. The header section provides general details such as file/format name, record length, and so forth. This information is common to all list items.

# Generic header format 0100

IBM i

Offset			
Dec	Hex	Type	Field
0	0	CHAR(64)	User area
64	40	BINARY(4)	Size of generic header
68	44	CHAR(4)	Structure's release and level
72	48	CHAR(8)	Format name
80	50	CHAR(10)	API used
90	5A	CHAR(13)	Data and time created
103	67	CHAR(1)	Information status
104	68	BINARY(4)	Size of user space used
108	6C	BINARY(4)	Offset to input parameter section
112	70	BINARY(4)	Size of input parameter section
116	74	BINARY(4)	Offset to header section
120	78	BINARY(4)	Size of header section
124	7C	BINARY(4)	Offset to list data section
128	80	BINARY(4)	Size of list data section
132	84	BINARY(4)	Number of list entries
136	88	BINARY(4)	Size of each entry
140	8C	BINARY(4)	CCSID of data in the list entries
144	90	CHAR(2)	Country or region ID
146	92	CHAR(3)	Language ID
149	95	CHAR(1)	Subseted list indicator
150	96	CHAR(42)	Reserved

© Copyright IBM Corporation 2011

Figure 3-37. Generic header format 0100

AS106.0

## Notes:

The above figure show the generic user space layout for the list APIs. There are several formats. Format 0100, which is shown, is the format for an original program model (OPM) layout. Format 0300 shows the format for an Integrated Language Environment (ILE) model layout. The fields are described in the information center in detail after the tables.

Note that source member QUSGEN, in QSYSINC/QRPGLESRC, can be used to define generic header data structure.

# Step 1: Create user space

IBM i

```

000003  D Message      C          'Unable to create User Space'
000005
000006  D SpaceName    S          20A  Inz('APISPACE  *CURLIB')
000007  D Attribute    S          10A  Inz('API_SPACE')
000008  D Size         S          10I 0 Inz(5000)
000009  D InitValue   S          1A   Inz('*')
000010  D Authority   S          10A  Inz('*USE')
000011  D Text         S          50A  Inz('API User Space')
000013  D Replace      S          10A  Inz('*YES')
000014  D ErrorCode    DS
000015  D BytesAvl    DS
000016  D BytesRet    DS
000017  D MsgId       7A
000018  D Reserved    1A
000019  D MsgDta      84A
000020
000021  D CreateUSpace PR          ExtPgm('QUSCRTUS')
000022  D
000023  D
000024  D
000025  D
000026  D
000027  D
000028  D
000029  D
000030
000031  /Free
000032  CallP CreateUSpace(SpaceName : Attribute : Size : InitValue : Authority :
000033           Text : Replace : ErrorCode);
000034
000042  If BytesRet <> 0;
000043    Disply Message '*EXT';
000044  Endif;
000045  *InLR = *On;
000047  /End-free

```

© Copyright IBM Corporation 2011

Figure 3-38. Step 1: Create user space

AS106.0

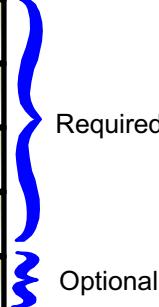
## Notes:

As we did earlier, we called the QUSCRTUS to create a user space named APISPACE.

## Example: List objects (QUSLOBJ) API parameters

IBM i

NAME	USE	TYPE
Qualified user space name	input	Char(20)
Format name	input	Char(8)
Qualified object & library name	input	Char(20)
Object type	input	Char(10)
Error code (optional)	i/o	Char(*)



Required

Optional

List Objects (QUSLOBJ) API is similar to the **DSPOBJD** CL command

© Copyright IBM Corporation 2011

Figure 3-39. Example: List objects (QUSLOBJ) API parameters

AS106.0

### Notes:

You can use the QUSLOBJ API to generate a list of object names and descriptive information based upon your specified selection parameters.

Once again, the qualified name contains the object and library names padded to 10 characters, with no / qualifier.

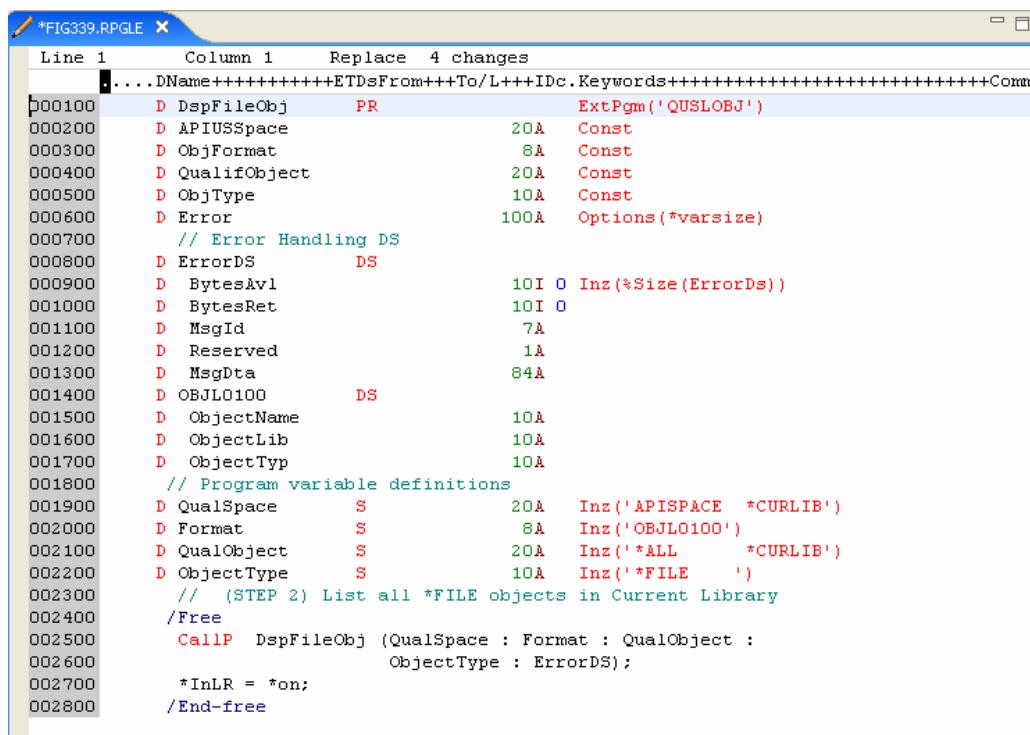
The object name may be explicit, generic, or **\*ALL**. The library name may be **\*ALL**, **\*ALLUSR**, **\*CURLIB**, **\*LIBL**, **\*USRLIBL**, or explicit.

The object type may be explicit, or **\*ALL**.

There are seven formats available for the output, ranging from OJL0100 (object names, the least information and the best performer), to OJL0700, (all object information and the most expensive performer).

## Step 2: Code program to fill user space

IBM i



```

*FIG339.RPGLE X
Line 1      Column 1      Replace 4 changes
. ....DName++++++ETDsFrom++To/L++IDc.Keywords+++++Comm
000100    D DspFileObj   PR           ExtPgm('QUSLOBJ')
000200    D APIUSpace        20A   Const
000300    D ObjFormat        8A   Const
000400    D QualifObject     20A   Const
000500    D ObjType         10A   Const
000600    D Error          100A  Options(*varszie)
000700    // Error Handling DS
000800    D ErrorDS        DS
000900    D BytesAvl        10I 0 Inz(%Size(ErrorDS))
001000    D BytesRet        10I 0
001100    D MsgId          7A
001200    D Reserved        1A
001300    D MsgData        84A
001400    D OBJL0100        DS
001500    D ObjectName       10A
001600    D ObjectLib        10A
001700    D ObjectTyp       10A
001800    // Program variable definitions
001900    D QualSpace        S           20A   Inz('APISPACE  *CURLIB')
002000    D Format           S           8A   Inz('OBJL0100')
002100    D QualObject       S           20A   Inz('*ALL      *CURLIB')
002200    D ObjectType       S           10A  Inz('*FILE    ')
002300    // (STEP 2) List all *FILE objects in Current Library
002400    /Free
002500    CallP  DspFileObj (QualSpace : Format : QualObject :
002600                           ObjectTyp : ErrorDS);
002700    *InLR = *on;
002800    /End-free

```

© Copyright IBM Corporation 2011

Figure 3-40. Step 2: Code program to fill user space

AS106.0

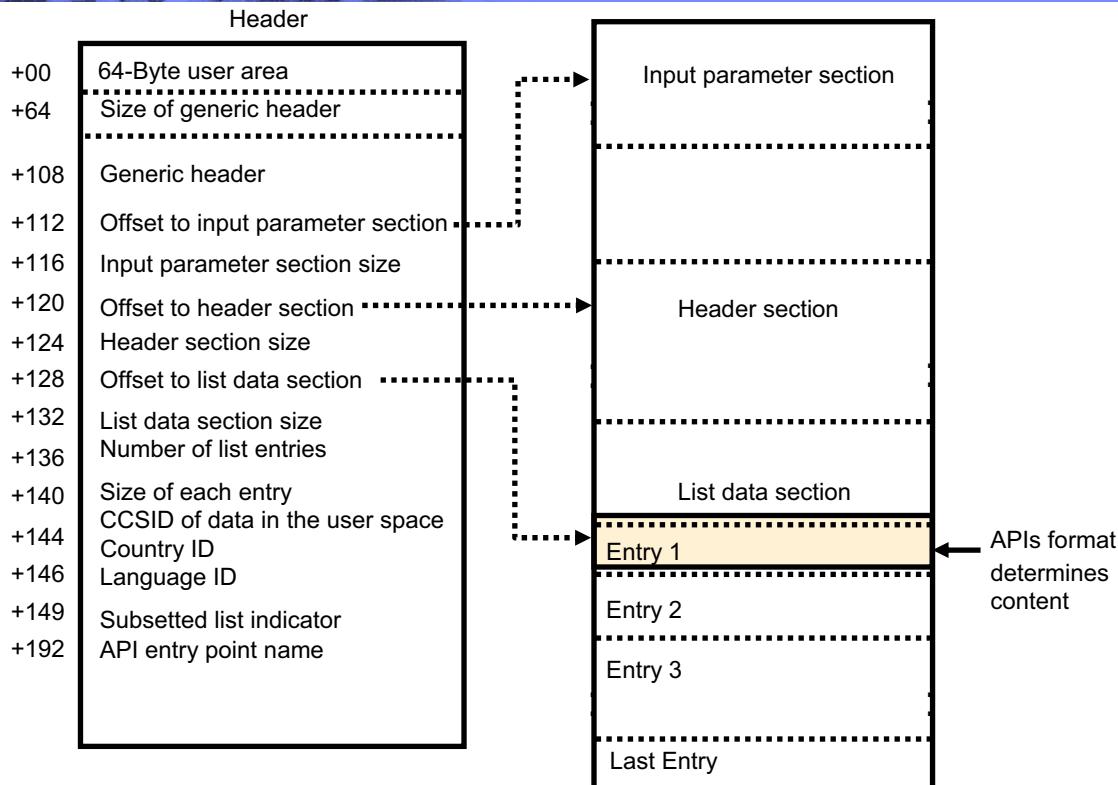
### Notes:

This program, which calls the QUSLOBJ List API, will load the user space with the requested information.

We use the format OBJL0100 to retrieve basic information and determine the format of the list entries that are output to the user space, APISPACE.

# Decimal offset

IBM i



© Copyright IBM Corporation 2011

Figure 3-41. Decimal offset

AS106.0

## Notes:

This visual shows you the layout of the data in the user space which was filled by the program in the previous visual.

# Format OBJL0100 for QUSLOBJ output

IBM i

Format of each list entry:

OFFSET	TYPE	FIELD
0	Char(10)	Object name used
10	Char(10)	Library name used
20	Char(10)	Object type used

© Copyright IBM Corporation 2011

Figure 3-42. Format OBJL0100 for QUSLOBJ output

AS106.0

## Notes:

# Parameters for retrieve user space API (QUSRTVUS)

IBM i

NAME	USE	TYPE
Qualified user space name	Input	Char(20)
Starting position	Input	Binary(4)
Length of data	Input	Binary(4)
Receiver variable	Output	Char(*)
Error code (optional)	I/O	Char(*)

© Copyright IBM Corporation 2011

Figure 3-43. Parameters for retrieve user space API (QUSRTVUS)

AS106.0

## Notes:

The Retrieve User Space (QUSRTVUS) API allows the contents of a user space to be extracted as a substring and placed in a single receiver parameter (consistent with all retrieve APIs).

# RPG IV program: Retrieve information from user space (1 of 3)

IBM i

```

FIG343.RPGLE X
Line 1      Column 1      Replace
.....1.....+.....2.....+.....3.....+.....4.....+.....5.....+.....6.....+.....7.....+.....8.....
000100      // Prototype for QUSRTVUS Control information
000200 >>1 D RtvUsrSpCtl    PR           ExtPgm('QUSRTVUS')
000300      D SpaceName          20A
000400      D StartPosition       10I 0
000500      D DataLength          10I 0
000600      D ReceiverVar        192A
000700      D Error              100A  Options(*varsize)
000800      // Error Handling DS
000900      D ErrorDS            DS
001000      D BytesAvl           10I 0  Inz(*Size(ErrorDS))
001100      D BytesRet            10I 0
001200      D MsgId              7A
001300      D Reserve             1A
001400      D MsgDta              84A
001500      // Prototype for QUSRTVUS Data Information
001600 >>2 D RtvUsrSpData   PR           ExtPgm('QUSRTVUS')
001700      D SpaceName          20A
001800      D StartPosition       10I 0
001900      D DataLength          10I 0
002000      D ReceiverVar        30A
002100      D Error              100A  Options(*varsize)

```

© Copyright IBM Corporation 2011

Figure 3-44. RPG IV program: Retrieve information from user space (1 of 3)

AS106.0

## Notes:

Now, we retrieve the object information from the user space. Remember that we wrote a list of all the objects in our course student master library.

This first visual shows the prototypes for the parameters to be passed to QUSRTVUS. There are two sets of parameters:

1. The control information
2. The actual information about each object to the user space

You see the difference in the receiver variables in the next visual.

# RPG IV program: Retrieve information from user space (2 of 3)

```

000121      // INPUT parameters for QUSRTVUS
000122 >>3  D QualSpace      S          20A   Inz('APISPACE  *CURLIB')
000123  D StartPos       S          10I  0
000124  D DataLength     S          10I  0
000125  D Count          S          Like(LstEntNo)
000126  D
000127  D Error          S          16A
000128      // OUTPUT parameters for QUSTRTVUS for return data
000129 >>4  D Receiver1    DS
000130  D UserArea        64
000131  D GenHdrSize     10I  0
000132  D RelLevel        4A
000133  D FormatUsed     8A
000134  D APIUsed         10A
000135  D CrtDatTim      13A
000136  D InfoStatus      1A
000137  D SizeSpace       10I  0
000138  D IPSSOffset     10I  0
000139  D IPSSize         10I  0
000140  D HdrOffset       10I  0
000141  D HdrSize         10I  0
000142  D LstOffset       10I  0
000143  D LstSize         10I  0
000144  D LstEntNo        10I  0
000145  D LstEntSize      10I  0
000146  D LstEntCCSD     10I  0
000147  D CountryId       2A
000148  D LanguageId     3A
000149  D SubsetInd       1A
000150  D Reserved         42A
000151  D
000152      // OUTPUT parameters - for return data (Format OBJL0100 List Entry)
000153 >>5  D Receiver2    DS
000154  D ObjName         10A
000155  D ObjLib          10A
000156  D ObjType         10A
000157  D

```

© Copyright IBM Corporation 2011

Figure 3-45. RPG IV program: Retrieve information from user space (2 of 3)

AS106.0

## Notes:

This visual contains three important areas:

3. The definition of the variables to be passed as parameters to QUSRTVUS. The program supplies the name of the user space (APISPACE) and the library in which it is located. The starting position is used to determine the starting location of the data that we are going to retrieve.
4. The receiver DS (Receiver1) that holds the control information to enable us to locate and retrieve the object data we loaded in the user space.
5. The receiver DS (Receiver2) that holds the object information. This structure is used to hold the information for each record written to the user space.

These structures are based on layouts found in QSYSINC/QRPGLESRC. Receiver1 is based on QUSGEN; Receiver2 is based on QUSLOBJ.

# RPG IV program: Retrieve information from user space (3 of 3)

IBM i

```

005900      // (STEP 3) Retrieve control information from User Space Generic Header
006000      /Free
006100 >>6   StartPos = 1;
006200     DataLength = %Size(Receiver1);
006300 >>7   CallP RtvUsrSpCtl (QualSpace : StartPos : DataLength : Receiver1
006400           : ErrorDS);
006500
006600      // (STEP 4) Retrieve list entries from User Space
006700 >>8   StartPos = LstOffset + 1;
006800     DataLength = %Size(Receiver2);
006900     Count = 1;
007000
007100 >>9   Dow Count <= LstEntNo;
007200 >>10    CallP RtvUsrSpData (QualSpace : StartPos : DataLength : Receiver2 :
007300           Error);
007400 >>11    StartPos = StartPos + LstEntSize;
007500     Count = Count + 1;
007600     Dspla ObjName '*REQUESTER';
007700   Enddo;
007800
007900     *InLR = *ON;
008000   /End-free

```

© Copyright IBM Corporation 2011

Figure 3-46. RPG IV program: Retrieve information from user space (3 of 3)

AS106.0

## Notes:

The logic of the program is very straightforward:

6. Initialize the *StartPos* variable to pick up the control information for the **Receiver1** DS. We also determine the size of the structure using the %Size BIF.
7. Call the QUSRTVUS API to retrieve the control information.
8. One piece of information that we use is *LstOffset* (part of the **Receiver1** DS). The value of *LstEntSize* enables us to determine the start of the output of the QUSLOBJ API.
9. We code a loop to process the list up to the number of list entries (**LstEntryNo**).
10. For each list entry, we call QUSRTVUS to get the entry.
11. We point to the next list entry by adding the value of *LstOffset* to *StartPos*.

# Retrieve pointer to user space (QUSPTRUS) API

IBM i

NAME	USE	TYPE
Qualified user space name	Input	Char(20)
Return pointer	Output	PTR(SPP)
Error code (optional)	I/O	Char(*)

© Copyright IBM Corporation 2011

Figure 3-47. Retrieve pointer to user space (QUSPTRUS) API

AS106.0

## Notes:

The Retrieve Pointer to User Space (QUSPTRUS) API retrieves a pointer to the contents of a user-domain user space. The data in that user space then can be directly manipulated by high-level language programs that support pointers.

# RPG IV program: Retrieve information from a user space using a pointer

```

FIG211.RPGL FIG210.RPGL FIG346.RPGL X
Line 1 Column 40 Replace
.....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8....+
000100 // Parameters for QUSPTRUS
000200 >>1 D QualSpace S 20A Inz('APISPACE *CURLIB')
000300 D Ptr1 S *
000400 D Ptr2 S *
000500 D Count S Like(LstEntNo)
000600 // Decode of required elements of Generic Header
000700 >>2 D Receiver1 DS Based(Ptr1)
000800 D LstOffset 10I 0 Overlay(Receiver1:125)
000900 D LstSize 10I 0 Overlay(Receiver1:129)
001000 D LstEntNo 10I 0 Overlay(Receiver1:133)
001100 D LstEntSize 10I 0 Overlay(Receiver1:137)
001200 // Decode of Format OBJL0100
001300 >>3 D Receiver2 DS Based(Ptr2)
001400 D ObjName 10A
001500 D ObjLib 10A
001600 D ObjType 10A
001700
001800 >>4 D RtvUsrSpcAddr PR ExtPgm('QUSPTRUS')
001900 D 20A
002000 D *
002100
002200 /Free
002300 // Retrieve control information from User Space Generic Header
002400 >>5 CallP RtvUsrSpcAddr(QualSpace:Ptr1);
002500
002600 // Retrieve list entries from User Space
002700 >>6 Ptr2 = Ptr1 + LstOffset;
For Count = 1 To LstEntNo;
  Dsply ObjName '*REQUESTER';
  Ptr2 = Ptr2 + LstEntSize;
EndFor;
002800
002900 >>7
003000 *InLR = *ON;
Return;
003100 /End-Free
003200
003300
003400
003500

```

© Copyright IBM Corporation 2011

Figure 3-48. RPG IV program: Retrieve information from a user space using a pointer

AS106.0

## Notes:

In the previous visual, we showed you some unnecessarily complex coding using offsets and the QUSRTVUS API.

We can retrieve the contents of a user space in a much simpler fashion using pointers and the QUSPTRUS API instead. This figure shows you the retrieval of information being performed by this API.

The coding steps follow:

1. Define the program variables. **Ptr1** and **QualSpace** are parameters to the QUSPRTUS API.
2. The receiver DS (Receiver1) that holds the control information to enable us to locate and retrieve the object data we loaded in the user space.
3. The receiver DS (Receiver2) that holds the object information. This structure is used to hold the information for each record written to the user space.
4. The prototype definition for the QUSPTRUS API.

5. Call the QUSPTRUS API.
6. Using the pointer to the user space, calculate the value of the first list entry (**Ptr2**) in the space.
7. As we retrieve items from the list, we loop and increment the value of **Ptr2** by the size of each entry item in the user space.

## Machine exercise: Using list APIs

IBM i



© Copyright IBM Corporation 2011

Figure 3-49. Machine exercise: Using list APIs

AS106.0

### Notes:

Perform the machine exercise using *list APIs*.

## Checkpoint (1 of 2)

IBM i

1. True or False: Using the **DEFINE** parameter of the **CRTBNDRPG** or the **CRTRPGMOD** command is the same as coding the /**DEFINE** condition-name directive on the first line of the RPG IV source member.
  
2. When testing conditions with directives within a source member:
  - a. /**ELSEIF** and /**ELSE** are invalid outside of an /**IF** group
  - b. An /**IF** group can contain only one /**ELSE** directive
  - c. Every /**IF** must have a matching /**ENDIF**
  - d. All of the above
  
3. By specifying \_\_\_\_\_ as the data type for a stand a lone field, the field is identified as a pointer field.
  - a. P
  - b. B
  - c. \*
  - d. BASED

© Copyright IBM Corporation 2011

Figure 3-50. Checkpoint (1 of 2)

AS106.0

### Notes:

## Checkpoint (2 of 2)

IBM i

4. The \_\_\_\_\_ keyword is specified for a data structure or stand a lone field in order to create a pointer field with the same name specified as the keyword's parameter.
  - a. BASED
  - b. P
  - c. B
  - d. \*
5. True or False: Before a based data structure or stand a lone field can be used, the basing pointer must be assigned a valid address.
6. In order to use a list API, you must create a \_\_\_\_\_ to hold its output.
  - a. System heap
  - b. Data area
  - c. User space
  - d. Control header

© Copyright IBM Corporation 2011

Figure 3-51. Checkpoint (2 of 2)

AS106.0

### Notes:

## Unit summary

IBM i

Having completed this unit, you should be able to:

- Code RPG IV programs that include conditional directives
- Describe user spaces and how to access them using an API
- Describe the purpose of the pointer data type
- Use a pointer to access a data item in an RPG IV program
- Create storage outside an RPG IV program dynamically
- Use a pointer to access storage outside of your RPG IV program

© Copyright IBM Corporation 2011

Figure 3-52. Unit summary

AS106.0

### Notes:



# Unit 5. ILE CEE API programming

## What this unit is about

This unit describes the use of ILE Common Execution Environment (CEE) bindable APIs. While we do not discuss them all, you gain an appreciation of the APIs that are available and how they can be used to enhance your ILE RPG IV based applications.

## What you should be able to do

After completing this unit, you should be able to:

- Create RPG IV programs that call ILE CEE APIs
- Know where to find reference material

## How you will check your progress

Accountability:

- Machine exercise
- Checkpoint questions

## References

i Information Center:

<http://publib.boulder.ibm.com/infocenter/iseries/v7r1m0/>

Click **Programming>APIs>APIs by Category**. In the window on the right, click **ILE CEE**.

*ILE Concepts* reference in the i Information Center.

## Unit objectives

IBM i

After completing this unit, you should be able to:

- Create RPG IV programs that call ILE CEE APIs
- Know where to find reference material

© Copyright IBM Corporation 2011

Figure 4-1. Unit objectives

AS106.0

### **Notes:**

## 5.1. An introduction to CEE ILE APIs

# What do the ILE CEE APIs provide?

IBM i

- Wide range of functional areas, including:
  - Activation group and control flow APIs
  - Condition management APIs
  - Date and time APIs
  - Math APIs
  - Message services APIs
  - Program or procedure call APIs
  - Source debugger
  - Storage management APIs
- Contained in system-supplied service program
  - Accessed through QILE \*BNDDIR
  - Automatically bound as part of normal bind

© Copyright IBM Corporation 2011

Figure 4-2. What do the ILE CEE APIs provide?

AS106.0

## Notes:

There are many ILE CEE APIs that are provided with the IBM i system. These APIs are also known as *bindable APIs*, which means that they are *automatically bound* to your program that calls them. You do not have to bind them when you create your program object.

Some of the APIs available in the categories shown above in the visual are as follows:

### Activation group and control flow APIs

*Abnormal End (CEE4ABN)* abnormally ends the activation group containing the nearest control boundary.

*Find a Control Boundary (CEE4FCB)* searches the call stack for the nearest call stack entry that is a control boundary.

*Normal End (CEETREC)* initiates a normal ending of the activation group containing the nearest control boundary.

*Register Activation Group Exit Procedure (CEE4RAGE)* registers procedures that are called when an activation group ends.

*Register Call Stack Entry Termination User Exit Procedure (CEERTX)* registers a user-defined procedure that runs when the call stack entry for which it is registered is ended by anything other than a return to the caller.

*Unregister Call Stack Entry Termination User Exit Procedure (CEEUTX)* is used to unregister a user-defined procedure that was previously registered by the Register Call Stack Entry Termination User Exit Procedure (CEERTX) API.

### Condition management APIs

These APIs allow you to handle errors independent of high-level language-specific error handling. We learned how to use CEEHDLR in the previous unit. Some of the condition management APIs are as follows:

*Construct a Condition Token (CEENCOD)* is used to dynamically construct a 12-byte condition token.

*Move the Resume Cursor to a Return Point (CEEMRCR)* moves the resume cursor to a return point relative to the current handle cursor.

*Register a User-Written Condition Handler (CEEHDLR)* registers a user-written condition handler for the current call stack entry.

*Unregister a User-Written Condition Handler (CEEHDLU)* unregisters a user-written condition handler for the current call stack entry.

### Date and time APIs

Some of the date and time APIs are as follows:

*Calculate Day of Week from Lilian Date (CEEDYWK)* returns the day of the week as a number between 1 and 7.

*Convert Date to Lilian Format (CEEDAYS)* converts a string representing a date into a number representing the number of days since 14 October 1582.

*Convert Lilian Date to Character Format (CEEDATE)* formats a number representing a Lilian date.

*Get Current Local Time (CEELOCT)* returns the current local time in three formats: Lilian date (the number of days since 14 October 1582), Lilian timestamp (the number of seconds since 00:00:00 14 October 1582), and Gregorian character string (in the form 'YYYYMMDDHHMISS999').

*Query Century (CEEQCEN)* queries the century within which two-digit year values are assumed to lie.

*Return Default Date and Time Strings for Country or Region (CEEFDMDT)* returns the default date and time picture strings for the country or region specified in the country/region\_code parameter.

### Math APIs

Some of the Math APIs are as follows (the x refers to the data type being processed):

Absolute Function (CEESxABS)

Cosine (CEESxCOS)

Exponential Base e (CEESxEXP)

Exponentiation (CEESxXPx)

Factorial (CEE4SxFAC)

Logarithm Base 10 (CEESxLG1)

Nearest Whole Number (CEESxNWN)

Sine (CEESxSIN)

The following math API can be accessed individually:

Basic Random Number Generation (CEERAN0)

## **Message Services**

These APIs can be used to dispatch, format, obtain, retrieve, and store messages. Some of the message services APIs are as follows:

Dispatch a Message (CEEMOUT) dispatches a message string.

Get a Message (CEEMGET) retrieves a message and stores it in a buffer.

## **Program or procedure call APIs**

Some examples are as follows:

Get String Information (CEEGSI) retrieves string information about a parameter used in the call to this API.

Retrieve Operational Descriptor Information (CEEDOD) retrieves operational descriptor information about a parameter used in the call to this API.

Test for Omitted Argument (CEETSTA) is used to test for the presence or absence of anmissible argument.

## **Source debugger bindable APIs**

Some examples are:

Register a View of a Module (QteRegisterDebugView)

Remove a View of a Module (QteRemoveDebugView)

Retrieve the Attributes of the Source Debug Session (QteRetrieveDebugAttribute)

Retrieve the List of Modules and Views for a Program (QteRetrieveModuleViews)

## **Storage management APIs**

Some examples are as follows:

Create Heap (CEECCRHP)

Discard Heap (CEEDSHP)

Get Heap Storage (CEEGTST)

# ILE CEE data types

IBM i

ILE CEE Data Type	Description	RPG IV Data Type
CHAR, UCHAR, SCHAR	1-byte character	A with length 1
CHAR $n$	$n$ -bytes character	A with length $n$
INT2	2-byte integer (U)	I with length 5
UINT2	2-byte integer (U)	U with length 5
INT4	4-byte integer	I with length 10
UINT4	4-byte integer (U)	U with length 10
FLOAT4	4-byte single precision FP	F with length 4
FLOAT8	8-byte double precision FP	F with length 8
Pointer	Address pointer	*
vString	Variable length string	A with VARYING

Note: Review the ILE CEE documentation for others

© Copyright IBM Corporation 2011

Figure 4-3. ILE CEE data types

AS106.0

## Notes:

This table is a subset of the data type table in the i Information Center. To view the complete set of data types, go to:

<http://publib.boulder.ibm.com/infocenter/iseries/v7r1m0>

and in the left hand pane:

Programming>APIs>APIs by category>ILE CEE APIs>Data Type Definitions of ILE CEE

# Coding to call a bindable API

IBM i

```
000100      D CEExxxxx          PR                  EXTPROC ('CEExxxxx')
000101      D parm1 ...
000102      D ...
000103
000104
000105      CALLP CEExxxxx( parm1 : parm2 : ... ;
000106                      parmn : feedback);
000107
000108      /End-free
000109      /
000110
```

© Copyright IBM Corporation 2011

Figure 4-4. Coding to call a bindable API

AS106.0

## Notes:

You call the bindable APIs in the same way that you called other APIs, for example, in the previous unit.

In the visual:

- **CEExxxxx** is the name of the bindable API
- **parm1, parm2, ... parmn** are omissible or required parameters passed to or returned from the called API.
- **feedback** is an omissible feedback code that indicates the result of the bindable API.

**Note:** Bindable APIs can be used only in an ILE activation group. If you create your program with the **CRTBNDRPG** command, you must specify **DFTACTGRP(\*NO)**.

## CEEGPID example: Parameters

IBM i

Parameter	Use	Type
1. CDD_Version	Output	INT4
2. Plat_ID	Output	INT4
<b>Omissible parameter:</b>		
3. fc	Output	FEEDBACK

© Copyright IBM Corporation 2011

Figure 4-5. CEEGPID example: Parameters

AS106.0

### Notes:

The Retrieve ILE Version and Platform ID (CEEGPID) API retrieves the ILE version ID and the platform (that is, operating system) ID. The IDs are those currently in use for processing the active condition.

### Required parameter group

**CEE\_Version** (output): A 32-bit numeric representation of the version of ILE that created this condition. For example, if 540 is returned as the version, it represents Version 5 Release 4 Modification 0.

**Plat\_ID** (output): A 32-bit numeric representation of the operating system on which this condition was created. The possible value for IBM i is 4.

### Omissible parameter

**fc** (output): A 12-byte feedback code.

- CEE0000 The API completed successfully
- CEE9902 Unexpected user error occurred in &1

# CEEgid: Retrieve current platform and release level


IBM i

```

000100      H DFTACTGRP (*NO)
000200
000300      D VerRel          PR           EXTPROC('CEEgid')
000400      D Version         10I 0
000500      D OSPlatform       10I 0
000600      D FeedBack        12A  Options(*Omit)
000700      D
000800      D Ver             S           10I 0
000900      D OS              S           10I 0
001000      D OpSys           S           6A
001100      D Msg             S           40A
001200
001300      /Free
001400      CALLP VerRel(Ver : OS: *Omit);
001500      If OS = 4;
001600          OpSys = 'OS/400';
001700
001800          Msg = 'You are an ' + OpSys + ' system at '
001900          + 'V' + %subst(%char(ver):1:1)
002000          + 'R' + %subst(%char(ver):2:1);
002100          DispMsg '*REQUESTER';
002200
002300      Else;
002400          Msg = 'We have an error since platform = ' + %char(OS);
002500      EndIf;
002600
002700      *inLR = *on;
002800      /End-free

```

© Copyright IBM Corporation 2011

Figure 4-6. CEEgid: Retrieve current platform and release level

AS106.0

## Notes:

The Retrieve ILE Version and Platform ID (CEEgid) API retrieves the ILE version ID and the platform (that is, operating system) ID. This API is one of the set of condition management APIs.

The guidelines that we discussed in unit 2 for the system APIs apply to the CEE APIs also.

Again, for the CEEgid API, the parameters are:

- Required:
  - a. **CEE\_Version**, Output, INT4 (4 bytes)
  - b. **Plat\_ID**, Output, INT4 (4 bytes)
- Omissible Parameter:
  - c. **fc** Output, FeedBack (12 bytes)

In our example above, the two parameters that are INT4 are declared as RPG IV 10I. The optional feedback parameter is omitted.

The first parameter, **Version**, contains left justified digits. For example, a value of 440 means this is a V4R4 system.

The second parameter, **OSPlatform**, also contains left-justified digits and equals **4** for an IBM i or OS/400 Operating System.

This API is contained in the system service program, QLEAWI, which is automatically bound to your RPG IV program.

# Operational descriptors

IBM i

- Offer flexibility in parameter passing
- Parameter can be different depending on OpDesc
- Used by some CEE APIs
- Specified on D-spec of procedure prototype

© Copyright IBM Corporation 2011

Figure 4-7. Operational descriptors

AS106.0

## Notes:

Sometimes you may want to pass a parameter to a procedure even though the data type is not precisely known to the called procedure. For example, when a parameter is able to accept different types of strings, you must specify operational descriptors on the call.

You use operational descriptors to provide descriptive information to the called procedure regarding the form of the parameter. The additional information allows the procedure to properly interpret the string. You should only use operational descriptors when they are expected by the called procedure.

You can request operational descriptors for both prototyped and non-prototyped parameters. For the prototyped CALLP, you specify the keyword OPDESC on the prototype definition.

Operational descriptors are then built by the calling procedure and passed as hidden parameters to the called procedure.

# CEEDAYS: Get Lilian date parameters

IBM i

## CEEDAYS: Get Lilian date parameters

1. **input\_char\_date** (Input) VSTRING
2. **picture\_string** (Input) VSTRING
3. **output\_Lilian\_date** (Output) INT4

Omissible:

4. **fc** (Output) FEEDBACK

```
CALLP CEEDAYS('6/2/88' , 'MM/DD/YY', lildate, fc);
CALLP CEEDAYS('06/02/88', 'MM/DD/YY', lildate, fc);
CALLP CEEDAYS('060288' , 'MMDDYY' , lildate, fc);
CALLP CEEDAYS('88154'   , 'YYDDD'   , lildate, fc);
```

- All assign the same value to variable *lildate*

© Copyright IBM Corporation 2011

Figure 4-8. CEEDAYS: Get Lilian date parameters

AS106.0

### Notes:

The Convert Date to Lilian Format (CEEDAYS) API converts a string representing a date into a number representing the number of days since October 14, 1582.

There are four parameters:

1. **input\_char\_date** (input by *descriptor*): A character string representing a date or timestamp in the format shown by parameter two, **picture\_string**. Field length can range from 5 to 255 characters. **Input-char-date** can contain leading or trailing blanks. Parsing for a date begins with the first non-blank character unless the picture string contains leading blanks, in which case CEEDAYS skips exactly that many positions before parsing begins. After a valid date is parsed, remaining characters are ignored. Valid dates are in the range October 15, 1582 to December 31, 9999.
2. **picture\_string** (input by *descriptor*): A character string indicating the format of the date value in **input\_char\_date**, for example, MM/DD/YY. Each character in **picture\_string** represents a character in **input\_char\_date**. If delimiters such as the slash (/) appear in the picture string, then leading zeros can be omitted. Note the examples in the visual all produce the same Lilian date.

If **picture\_string** is null or blank, CEEDAYS obtains a value for this second parameter based on the current job value for the country or region ID (CNTRYID). For example, if the current value for CNTRYID is US (United States), the date format is MM/DD/YY. If the current job value for CNTRYID is FR (France), the date format is DD.MM.YYYY.

3. **output\_Lilian\_date** (output): A 32-bit binary integer representing the Lilian date, which is the number of days since October 14, 1582. For example, May 16, 1988 is day number 148138. If **input\_char\_date** does not contain a valid date, **output\_Lilian\_date** is set to 0 and CEEDAYS ends with a nonzero feedback code.
4. **fc** (output): A 12-byte feedback code passed by reference. If specified as an argument, feedback information (a condition token) is returned to the calling procedure. If not specified and the requested operation was not successfully completed, the condition is signaled to the condition manager. The documentation includes a list of the feedback code values.

The feedback parameter is best described in a data structure. One of the subfields is the severity code. If the severity is zero, the API completed successfully.

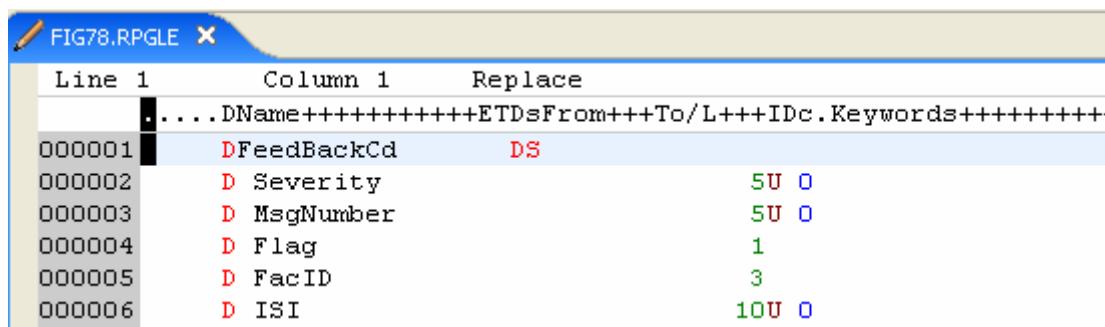
If you omit the feedback code parameter when you are calling an ILE CEE API, and the API fails, the API sends an exception message to the caller.

CEEDAYS is one of the Date and Time CEE APIs.

## ILE feedback code

IBM i

- Also known as condition token
- Used in condition handler



The screenshot shows a table titled 'FIG78.RPGLE' with columns 'Line 1', 'Column 1', and 'Replace'. The table lists six rows of data:

Line 1	Column 1	Replace
000001	DFeedBackCd	DS
000002	D Severity	5U O
000003	D MsgNumber	5U O
000004	D Flag	1
000005	D FacID	3
000006	D ISI	10U O

© Copyright IBM Corporation 2011

Figure 4-9. ILE feedback code

AS106.0

### Notes:

As input to an ILE CEE API, you have the option of coding a **feedback code** and using the it as a return (or feedback) code check in a procedure. The feedback code is a condition token value that is provided for flexibility in checking returns from calls to other procedures. You can then use the feedback code as input to a condition token. You will use a condition token when we work with condition handlers in an upcoming unit.

In RPG IV, you should code the feedback code as a data structure as shown in the visual.

ILE condition handling includes the following functions:

- Ability to dynamically register an ILE condition handler
- Ability to signal an ILE condition
- Condition token architecture
- Optional condition token feedback codes for bindable ILE APIs

The subfields of the feedback code data structure are:

**Severity:** A three-bit binary integer that indicates the severity of the condition.

**Msg\_No:** A two-byte binary number that identifies the message associated with the condition. The combination of **Facility\_ID** and **Msg\_No** (described below) uniquely identifies a condition.

**Flag:** A two-bit field that defines the format of the Condition\_ID portion of the token. ILE conditions are always **1**.

**Facility\_ID:** A three-character alphanumeric string that identifies the facility that generated the condition. The **Facility\_ID** indicates whether the message was generated by the system or a high-level language (HLL) run time. Some examples of facility ids are:

**CEE:** ILE common library

**CPF:** OS/400 CPF message

**MCH:** OS/400 Machine Exception

**RNX:** RPG IV Exception

**I\_S\_Info:** A four-byte field that identifies the instance specific information associated with a given instance of the condition. This field contains the reference key to the instance of the message associated with the condition token. If the message reference key is zero, there is no associated message.

If you code the feedback code parameter in your application to receive feedback information from an ILE CEE API, the following sequence of events occurs when a condition is raised:

1. An informational message is sent to the caller of the API, communicating the message associated with the condition.
2. The ILE CEE API in which the condition occurred builds a condition token for the condition. The ILE CEE API places information into the instance specific information area. The instance specific information of the condition token is the message reference key of the informational message. This is used by the system to react to the condition.
3. If a detected condition is critical (severity is 4), the system sends an exception message to the caller of the ILE CEE API.
4. If a detected condition is not critical (severity less than 4), the condition token is returned to the routine that called the ILE CEE API.

5. When the condition token is returned to your application, you can:
  - Ignore it and continue processing.
  - Signal the condition using the Signal a Condition (CEESGL) bindable API.
  - Get, format, and dispatch the message for display using the Get, Format, and Dispatch a Message (CEEMSG) bindable API.
  - Store the message in a storage area using the Get a Message (CEEMGET) bindable API.
  - Use the Dispatch a Message (CEEMOUT) bindable API to dispatch a user-defined message to a destination that you specify.
  - When the caller of the API regains control, the informational message is removed and does not appear in the job log.

If you omit the feedback code parameter when you are calling an ILE CEE API, the ILE CEE API sends an exception message to the caller of the bindable API if a condition is raised.

In addition to the Information Center, you should review *Condition Token Testing* in *Chapter 9: Exception and Condition Management* of the *ILE Concepts* reference.

# CEEDAYS example (1 of 2)

IBM i

```

000100      H DFTACTGRP (*NO)
000200
000300      // Prototype for CEEDAYS
000400 >>1  D GetLilianDte    pr          ExtProc('CEEDAYS')
000500 >>2  D                         Opdesc
000600      D                         10A  Varying
000700      D                         10A  Varying
000800      D                         10I  0
000900      D                         12A  Options(*Omit)
001000      // Definitions
001100      D Today           S          10A  Varying
001200      D January1        S          10A  Inz('2002-01-01')
001300      D
001400      D DateFmt         S          10A  INZ('YYYY-MM-DD')
001500      D
001600      D LDDayNo        S          10I  0
001700      D LDDayJan1     S          Like(LDDayNo)
001800
001900 >>3  D FeedBack        DS
002000      D Severity         S          5U  0
002100      D Msgno           S          5U  0
002200      D Flag             1
002300      D FacilityId     3
002400      D Isi              10U  0
002500
002600      D NoDays          S          4  0
002700      D ErrorText       S          30A  Inz('Feedback Code: ') Varying
002800

```

© Copyright IBM Corporation 2011

Figure 4-10. CEEDAYS example (1 of 2)

AS106.0

## Notes:

This is the first of two visuals that show you an example of using CEEDAYS. We use our example to calculate the difference in days between two Lilian dates, January 1, 2002 and today.

In this visual, we are defining the prototype for the CEEDAYS API and stand-alone fields. Notice:

1. The prototype for CEEDAYS.
2. The keyword OPDESC as required by the API. If you do not include this keyword, the API fails.
3. A data structure is used to define the feedback parameter.

## CEEDAYS example (2 of 2)

IBM i

```

002900 // Main Procedure
003000
003100 /FREE
003200 >>4 Today = %char(%date(*date));
003300             Dsply Today '*REQUESTER';
003400 >>5 CallP GetLilianDte(Today : DateFmt : LDDayNo : FeedBack);
003500
003600     ExSr CheckFB;
003700
003800     Dsply LDDayNo '*REQUESTER';
003900
004000             Dsply January1 '*REQUESTER';
004100 >>6 CallP GetLilianDte(January1 : DateFmt : LDDayJan1 : FeedBack);
004200
004300     ExSR CheckFB;
004400
004500             Dsply LDDayJan1 '*REQUESTER';
004600
004700 // Calculate duration
004800 >>7 NoDays = LDDayNo - LDDayJan1;
004900             Dsply NoDays '*REQUESTER';
005000
005100 *INLR = *ON;
005200
005300 >>8 BegSR CheckFB;
005400     If Severity <> 0;
005500         ErrorText = ErrorText + FeedBack;
005600             Dsply ErrorText '*REQUESTER';
005700         Return;
005800         *InLR = *on;
005900     Endif;
006000 EndSR;
006100
006200 /END-FREE

```

© Copyright IBM Corporation 2011

Figure 4-11. CEEDAYS example (2 of 2)

AS106.0

### Notes:

This second visual shows you the logic of the program:

4. Get the value of today's date and put it in character format. Since we did not specify a date format, it is **\*ISO** by default.
5. Call CEEDAYS to obtain the Lilian equivalent of today's date.
6. Call CEEDAYS again to obtain the Lilian date for January 1, 2002.
7. Determine the number of days since January 1, 2002.
8. If there is a problem with the feedback DS (that is severity not equal to 0) received from CEEDAYS, we simply issue a message and quit.

# CEELOCT: Get current local time parameters

IBM i

Required:

1. **output\_Lilian** (Output) INT4
2. **output\_seconds** (Output) FLOAT8
3. **output\_Gregorian** (Output) CHAR23

Omissible:

4. **fc** (Output) FEEDBACK

**Extract current local date and time in the form  
YYYYMMDDHHMISS999:**

```
CALLP CEELOCT (days, secs, localdatetime, fc);
```

© Copyright IBM Corporation 2011

Figure 4-12. CEELOCT: Get current local time parameters

AS106.0

## Notes:

The Get Current Local Time (CEELOCT) API returns the current local time in three formats: Lilian date (the number of days since October 14, 1582), Lilian timestamp (the number of seconds since 00:00:00 October 14, 1582), and Gregorian character string (in the form 'YYYYMMDDHHMISS999'). These values are compatible with the other ILE date and time APIs and with existing language intrinsic functions.

The parameters are:

1. **output\_Lilian** (output): A 32-bit binary integer representing the current local date in the Lilian format.
2. **output\_seconds** (output): A 64-bit double floating point number representing the current local date and time as the number of seconds since 00:00:00 on October 14, 1582.
3. **output\_Gregorian** (output): A 17-byte character string in the form 'YYYYMMDDHHMISS999' representing local year, month, day, hour, minute, second, and millisecond.

4. **fc** (output): A 12-byte feedback code passed by reference. If specified as an argument, feedback information (a condition token) is returned to the calling procedure. If not specified and the requested operation was not successfully completed, the condition is signaled to the condition manager. As always, if the severity subfield is zero, the API completed successfully.

# CEELOCT example (1 of 2)

IBM i

```

000100      H DFTACTGRP (*NO)
000200
000300          // Prototype for CEE_DAYS
000400 >>1  D GetLocalTime    PR           ExtProc('CEELOCT')
000600      D                                         10I 0
000700      D                                         8F
000800      D                                         20A
000900      D                                         12A   Options(*Omit)
001000          // Definitions
001100      D LilianInt      S             10I 0
001101      D FloatDtTime    S             8F
001102 >>2  D CharTimeStamp  S             20A
001501
001600      D FeedBack       DS
001700      D Severity        S             5U 0
001800      D Msgno          S             5U 0
001900      D Flag            S             1
002000      D FacilityId    S             3
002100      D Isi             S             10U 0
002200
002600      D ErrorText      S             30A   Inz('Feedback Code: ') Varying
002601 >>3  D TimeStamp      S             Z
002602      D CharFloatDtTm S             50A   Varying
002603      D CharLilianInt S             Like(CharFloatDtTm)

```

© Copyright IBM Corporation 2011

Figure 4-13. CEELOCT example (1 of 2)

AS106.0

## Notes:

This API outputs four parameters, as you have seen. In this visual, you can see the definitions of the parameters, the prototype, and some other fields. Notice:

1. The prototype has no input parameters. All parameters are returned by the API. In this example, we are most interested in the third parameter, which returns a 23-character value representing the current timestamp.
2. This is the character representation of the timestamp. The length in this program is 20 characters, which is compatible with the length of the IBM i timestamp data type.
3. In this example, we are going to use the CEELOCT API to place a value in a *timestamp* (Z data type) variable. We define the variable.

## CEELOCT example (2 of 2)

IBM i

```

003000      /FREE
003500 >>4   Call1P GetLocalTime(LilianInt : FloatDtTime : CharTimeStamp : FeedBack);
003600
003700     ExSr CheckFB;
003701     // Convert to character for Dsply
003702     CharLilianInt = %char(LilianInt);
003800     CharFloatDtTm = %char(FloatDtTime);
003802
003900     Dsply CharLilianInt '*REQUESTER';
003901     Dsply CharFloatDtTm '*REQUESTER';
004300     Dsply CharTimeStamp '*REQUESTER';
004301     // Convert character format to Timestamp; trunc last three characters
004303 >>5   TimeStamp = %Timestamp(ChartimeStamp : *ISO0);
005500
005600     *INLR = *ON;
005700
005800 >>6   BegSR CheckFB;
005900     If Severity <> 0;
006000     ErrorText = ErrorText + FeedBack;
006100     Dsply ErrorText '*REQUESTER';
006200     Return;
006300     *InLR = *on;
006400     Endif;
006500     EndSR;
006600
006700 /END-FREE

```

© Copyright IBM Corporation 2011

Figure 4-14. CEELOCT example (2 of 2)

AS106.0

### Notes:

CEELOCT is an excellent alternative to using the TIME opcode, which is not supported in free format.

This visual shows you the logic of the procedure:

4. The API is called.
5. The character parameter is converted to a true timestamp data type.
6. The procedure checks whether the API severity return code is non-zero and returns control to the caller of the procedure if it is.

## Math functions

IBM i

- Called through CALLP or BIF-like function call
- Support integer and floating point numeric data types
- Specify parameter type in fifth character of API name:
  - CEE5IABS: Determines absolute value of integer
  - CEE4SSFAC: Calculates factorial of floating point number

© Copyright IBM Corporation 2011

---

Figure 4-15. Math functions

AS106.0

### Notes:

## CEE4SxFAC: N factorial

IBM i

- Calculates factorial of I = short integer (10 position), J = long integer (20 position)
- I = INT4 = RPG IV 10I 0
- J = INT8 = RPG IV 20I 0
- Formula is  $n! = 1*2*3 \dots (n-1)*n.$
- Required:
  - 1. **parm1** (Input) Integer I or J data type
  - 2. **result** (Output) Integer I or J data type
- Omissible:
  - 3. **fc** (Output) FEEDBACK
- CallP CEE4SIFAC (NumInteger : NFactor : FeedBack);

© Copyright IBM Corporation 2011

Figure 4-16. CEE4SxFAC: N factorial

AS106.0

### Notes:

This API calculates the factorial of an integer. The parameters can be a short or a long integer but they must both be the same length and must match the fifth character of the name of the API.

For the math APIs, the fifth character provides information about the data type being used. In the case of the CEE4SxFAC API, the *x* can be a short integer (RPG IC 10I 0) or a long integer (RPG IV 20I 0).

To find out about the parameters and data types for the math APIs, go to the i Information Center and follow this link:

<http://publib.boulder.ibm.com/infocenter/iseries/v7r1m0/>

Programming > APIs > APIs by category > ILE CEE > Math APIs >  
Calling Math Bindable APIs

Understanding the parameters and how they are used is critical to successful use of the Math APIs.

# CEE4SIFAC example

IBM i

```

000100  H DFTACTGRP (*NO)
000200
000300      // Prototype for CEE4SIFAC
000400 >>1  D NFactorial      PR          ExtProc('CEE4SIFAC')
000600  D           10I 0
000700  D           10I 0
000900  D           12A  Options(*Omit)
001000  // Definitions
001100 >>2  D NumInteger      S           10I 0 Inz(12)
001200  D NFactor        S           10I 0
001600 >>3  D FeedBack       DS
001700  D Severity        5U 0
001800  D Msgno          5U 0
001900  D Flag            1
002000  D FacilityId     3
002100  D Isi             10U 0
002101  D Message         40A
002200
002600  D ErrorText       S           30A  Inz('Feedback Code: ') Varying
002700
002900  // Main Procedure
003000 /FREE
003500 >>4  CallP NFactorial (NumInteger : NFactor : FeedBack);
003600
003700  ExSr CheckFB;
003701  Message = %char(NumInteger) + ' factorial = ' + %char(NFactor);
003900  Disply Message '*REQUESTER';
003901
005600  *INLR = *ON;
005700
005800 >>5  BegSR CheckFB;
005900  If Severity <> 0;
006000  ErrorText = ErrorText + FeedBack;
006100  Disply ErrorText '*REQUESTER';
006200  Return;
006300  *InLR = *on;
006400  Endif;
006500  EndSR;
006600
006700 /END-FREE

```

12 factorial = 479001600

© Copyright IBM Corporation 2011

Figure 4-17. CEE4SIFAC example

AS106.0

## Notes:

This is an example of calculating the factorial of an integer using the CEE4SIFAC API. The *I* in position 5 of the API name means that the API expects **10I 0** (10 position integer, no decimal positions) integer parameters for input and output.

Notice in the visual:

1. The prototype for the API. Note the *I* in position 5 of the API name. Also, notice that the integers are **10I 0** (10 position integer, no decimal positions) and that they match the fifth character of the API name for type.
2. The variables used in the program, where *NumInteger* is the number for which we will calculate the factorial. **NFactor** will hold the result.
3. The feedback DS.
4. API is called and displayed.
5. Subroutine to check the severity code. As with other CEE APIs, this error checking is the same as you have seen before.

## CEESxMOD: Modular arithmetic

IBM i

- Calculates remainder of  $n/m$
- $m$  must not be zero
- Parms can be integer (INT4 or INT8) or floating point
- Required:
  - 1. **parm1** (Input) Integer I or J Data type (or float)
  - 2. **parm2** (Input) Integer I or J Data type (or float)
  - 3. **result** (Output) Integer I or J Data type (or float)
- Omissible:
  - 4. **fc** (Output) FEEDBACK

© Copyright IBM Corporation 2011

Figure 4-18. CEESxMOD: Modular arithmetic

AS106.0

### Notes:

The CEESxMOD API calculates the remainder of  $n/m$  where  $m$  is non zero. Like the CEExFAC API, the parameters must match the fifth character of the name of the API. In this API,  $x$  can be **I** (10I 0) or **J** (20I 0) for an integer calculation. You can also specify **S** or **D** for floating point variables.

Three parameters are required. You must pass the dividend and the divisor as the first two parameters. The third parameter is the remainder of the division.

# CEESIMOD example

```

000100      H DFTACTGRP(*NO)
000200
000300      // Prototype for CEEDAYS
000400 >>1  D FRemainder    PR          ExtProc('CEESIMOD')
000500     D                                     10I 0
000600     D                                     10I 0
000700     D                                     10I 0
000800     D                                     12A  Options(*Omit)
000900
001000 // Definitions
001100 >>2  D NumInteger   S           10I 0 Inz(100)
001200     D Divisor       S           10I 0 Inz(6)
001201     D Remainder     S           10I 0
001300     D Feedback       DS
001400 >>3  D Severity      S           5U 0
001500     D Msgno        S           5U 0
001600     D Flag          I           1
001700     D FacilityId   S           3
001800     D Isi           10U 0
001900     D Message        40A
002000
002100     D ErrorText     S           30A  Inz('Feedback Code: ') Varying
002200
002300 // Main Procedure
002400
002500 /FREE
002600 >>4  CallIP FRemainder (NumInteger : Divisor : Remainder: FeedBack);
002700
002800     ExSr CheckFB;
002900     Message = 'Remainder of ' + %char(NumInteger)
003000     + ' / ' + %char(Divisor) + ' = '
003100     + %char(Remainder);
003200     Dsplay Message '*REQUESTER';
003300
003400     *INLR = *ON;
003500
003600 >>5  BegSR CheckFB;
003700     If Severity <> 0;
003800       ErrorText = ErrorText + FeedBack;
003900       Dsplay ErrorText '*REQUESTER';
00400     Return;
004100     *INLR = *on;
004200   Endif;
004300   EndSR;
004400
004500 /END-FREE

```

Remainder of 100 / 6 = 4

Figure 4-19. CEESIMOD example

AS106.0

## Notes:

In this example, we are dividing **NumInteger** by **Divisor**. The remainder is returned in **Remainder**. In the visual:

1. We define the prototype. Notice that the integer variables are each defined in a consistent manner with the name CEESIMOD.
2. We define the stand alone variables for the procedure, consistent with the prototype.
3. The feedback DS is what we have been using for all math APIs.
4. The API is called and the remainder is displayed.
5. The feedback DS is checked for any problems.

## CEETSTA: Test arguments

IBM i

- Test for the presence or absence of an omissible argument
- Required:
  - 1. **presence\_flag** (Output) INT4
  - 2. **arg\_num** (Input) INT4
- Omissible:
  - 3. **fc** (Output) FEEDBACK

© Copyright IBM Corporation 2011

Figure 4-20. CEETSTA: Test arguments

AS106.0

### Notes:

The CEETSTA API is used in the called procedure to test for the presence or absence of an omissible parameter. The first parameter is an output parameter, where a value of **1** means that the parameter was passed. A value of **0** means that the parameter was omitted.

The second parameter is the argument number that you want to test for presence.

Both parameters are defined as RPG IV **10I 0**.

As with other APIs, the feedback parameter is omissible. If you use it, a severity code of zero means the API was successful.

# CEETSTA example

IBM i

```

000001      H DftActGrp (*No)
000002
000003 >>1  D TestParms      PR          ExtProc('CEETSTA')
000004      D Parm_Passed    PR          10I 0
000005      D Parm_Number    PR          10I 0 Const
000006      D FeedBack       PR          12A Options(*Omit)
000007
000008 >>2  D ParmPres       PR          ExtPgm('PARMPRES')
000009      D                   PR          6   0
000010      D                   PR          25A
000011 >>3  D                   PR          10A Options(*Omit)
000012      D
000013      // Procedure Interface - 3 parms expected
000014 >>4  D ParmPres       PI          6   0
000015      D Parm1           PI          25A
000016      D Parm2           PI          10A Options(*Omit)
000017 >>3  D Parm3           PI          10A Options(*Omit)
000018
000019      D Parm_Passed    S           10I 0
000020      D Parm_Number    S           10I 0
000021
000022      /Free
000023 >>5  Parm_Number = 3; //Test whether Parm 3 passed or omitted
000024      Exsr Test_Parm;
000025      If Parm_Passed = 1;
000026      // More code
000027      EndIf;
000028
000029      *InLr = *On;
000030      Return;
000031
000032      // Test whether ommissible parm was passed
000033      Begsr Test_Parm;
000034 >>6  CallP TestParms ( Parm_Passed : Parm_Number : *Omit);
000035      EndSr;
000036
000037      /End-Free

```

© Copyright IBM Corporation 2011

Figure 4-21. CEETSTA example

AS106.0

## Notes:

Let us assume that we have a procedure that expects three parameters. The third parameter can be omitted. We would like to find out whether it was or was not passed. To do this, take note of:

1. The prototype for the CEETSTA API.
2. The prototype for this procedure, named TestParms.
3. The *third parameter* that is passed to TestParms, which *may be omitted*.
4. The procedure interface for TestParms.
5. The value of the parameter we want to test(set to 3). Our code could obviously be much more robust. In the If group, we check whether the parameter was passed by checking the value of **Parm\_Passed** (returned by CEETSTA).
6. The call to CEETSTA. In this example, we are not passing the feedback parameter. It could be easily added.

## \*PGM and \*SRVPGM information

IBM i

### List ILE Program Information (QBNLPGMI) API

Required Parameter Group:

1	Qualified user space name	Input	Char(20)
2	Format name	Input	Char(8)
3	Qualified ILE program name	Input	Char(20)
4	Error Code	I/O	Char(*)

Default Public Authority: \*USE

Threadsafe: No

### List Service Program Information (QBNLSPGM) API

Required Parameter Group:

1	Qualified user space name	Input	Char(20)
2	Format name	Input	Char(8)
3	Qualified service program name	Input	Char(20)
4	Error Code	I/O	Char(*)

Default Public Authority: \*USE

Threadsafe: No

© Copyright IBM Corporation 2011

Figure 4-22. \*PGM and \*SRVPGM information

AS106.0

### Notes:

These two APIs deserve to be mentioned at this point, even though they are not ILE CEE APIs.

The List ILE Program Information (QBNLPGMI) API gives information about ILE programs, similar to the Display Program (**DSPPGM**) command. The information is placed in a user space specified by you.

If an original program model (OPM) program is specified for the qualified ILE program name, an error is returned and the user space is not changed.

You can use the QBNLPGMI API to:

- List modules bound into an ILE program
- List service programs bound to an ILE program
- List data items exported to the activation group
- List data item imports that are resolved by weak exports that were exported to the activation group
- List copyrights of an ILE program

One of six output formats may be specified.

The List Service Program Information (QBNLSPGM) API gives information about service programs, similar to the Display Service Program (**DSPSRVPGM**) command. The information is placed in a user space specified by you.

You can use the QBNLSPGM API to:

- List modules bound into a service program
- List service programs bound to a service program
- List data items exported to the activation group
- List data item imports that are resolved by weak exports that were exported to the activation group
- List copyrights of a service program
- List procedure export information of a service program
- List data export information of a service program
- List signatures of a service program

One of ten output formats may be specified.

## Machine exercise: Using bindable CEE APIs

IBM i



© Copyright IBM Corporation 2011

Figure 4-23. Machine exercise: Using bindable CEE APIs

AS106.0

### Notes:

Perform the machine exercise *using bindable CEE APIs*.

## Checkpoint

 IBM i

1. True or False: ILE CEE APIs are automatically bound to the program that calls them without explicit binding when you create your program object.
  
2. Operational descriptors allow:
  - a. Omission of passed parameters
  - b. Flexibility in parameter passing
  - c. A parameter to accept different types of strings
  - d. Parameters to be passed in any sequence
  
3. True or False: If you omit the feedback code parameter when you are calling an ILE CEE API, the API fails.

© Copyright IBM Corporation 2011

---

Figure 4-24. Checkpoint

AS106.0

### Notes:

## Unit summary

IBM i

Having completed this unit, you should be able to:

- Create RPG IV programs that call ILE CEE APIs
- Know where to find reference material

© Copyright IBM Corporation 2011

Figure 4-25. Unit summary

AS106.0

### Notes:



# Unit 6. Leveraging DB2 UDB database features

## What this unit is about

This unit describes commitment control, triggers, and their programming considerations and uses. RPG IV considerations and examples for programming triggers are discussed in detail. Also introduced is the use of embedded SQL to perform file I/O in RPG IV programs.

## What you should be able to do

After completing this unit, you should be able to:

- Describe the elements of commitment control
- Implement basic commitment control in an RPG IV application
- Describe how trigger programs can be used to uniformly enforce business rules regarding a database
- Activate a trigger program to a DB2 UDB file
- Code a trigger program, that is, a program to receive the trigger buffer from the system when a trigger program is fired
- Describe the purpose of embedded SQL
- Recognize embedded SQL in an RPG IV program

## How you will check your progress

Accountability:

- Machine exercise
- Checkpoint questions

## Unit objectives

IBM i

After completing this unit, you should be able to:

- Describe the elements of commitment control
- Implement basic commitment control in an RPG IV application
- Describe how trigger programs can be used to uniformly enforce business rules regarding a database
- Activate a trigger program to a DB2 UDB file
- Code a trigger program, that is, a program to receive the trigger buffer from the system when a trigger program is fired
- Describe the purpose of embedded SQL
- Recognize embedded SQL in an RPG IV program

© Copyright IBM Corporation 2011

Figure 5-1. Unit objectives

AS106.0

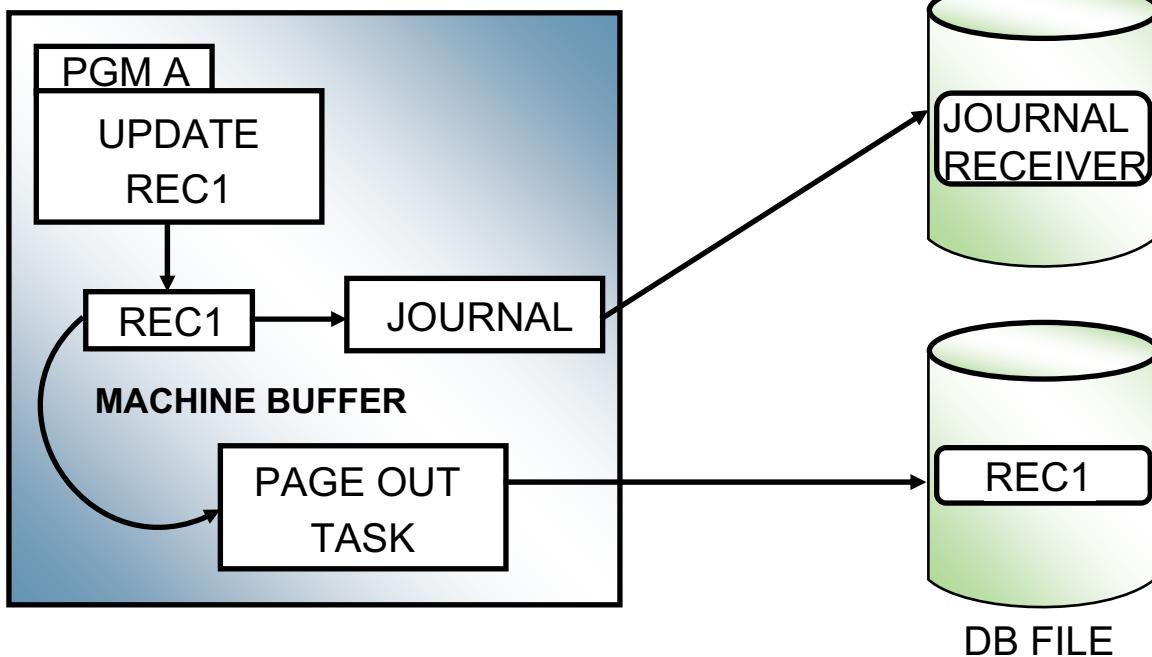
### Notes:

## 6.1. Commitment control

# DB journaling: Protection against loss

IBM i

## MAIN STORAGE



© Copyright IBM Corporation 2011

Figure 5-2. DB journaling: Protection against loss

AS106.0

## Notes:

When data records are to be written to disk, the system collects the records in a file buffer in main storage (memory) until the file buffer becomes full. Once that file buffer fills:

- Writing to DASD is independent of HLL program write
- Updated records may not be written for a while

Each write, update, or delete that is performed within an application program does not cause the changed record to be written to disk immediately. Changed records are moved from the job's record buffer (in the open data path) to the system buffer that is located in main storage. The system buffer can accumulate changes from multiple jobs using that file. The system performs a page out task that moves the data records to disk when the system buffer fills. This may be performed because the main storage is needed for other purposes, or some program using the file issues a close of the file.

- System buffer may be lost if:
  - ENDJOB occurs

- System ends abnormally
- ENDJOBABN is performed

If there are records in the open data path (ODP) buffer or the system buffer at times of abnormal end, these records may be lost.

- Journaling allows:
  - Recovery of database files using *captured* transactions
  - Commitment control implementation
  - Database file auditing

The presence of journaling in an application design protects against potential data loss. When the ODP buffer is passed to the system buffer, those record images are immediately queued to disk to be written to the journal receiver. While the journal receiver is written to immediately, the records are not necessarily written to the database at the same time. If the failure causes data in the ODP buffer or system buffer to be lost, the data has been captured in the journal receiver. It can be used to update the database without having to rekey the changes.

Journaling also protects against individual job failure by allowing recovery of files that are used by the job. This reduces the impact of the job failure on other users of the same application. For instance, if files have to be restored because of corruption, users do not need to rekey the transactions.

Journaling also provides an audit log of transaction updates. This is a legal requirement for some business sectors. Journal entries can be converted to a database file, which can be used to extract activity reports, audit trails, and security reports. Journalized data can also assist you to debug programs.

# Implementing journaling

IBM i

1. Create the journal receiver

```
CRTJRNRCV JRNRCV( )
```



2. Create the journal

```
CRTJRN JRN( ) JRNRCV( )
```



3. Begin the journal management process

```
STRJRNPF FILES( ) JRN( )
```



4. Save the files

© Copyright IBM Corporation 2011

Figure 5-3. Implementing journaling

AS106.0

## Notes:

Journaling is a file-oriented function that is provided by the system. Journaling can be started or ended very easily. It requires no additional programming or changes to existing programs and can be activated at the file level.

When a change is made to a file and you are using journaling, the system records the change in a journal receiver and writes the receiver to auxiliary storage before it is recorded in the file. Journal receivers are normally held in a different library and auxiliary storage pool than the journaled files.

The four steps outlined above will enable journaling for specific database files on the system. All database files updated during a defined transaction under commitment control must be journaled to the same journal.

# Journal codes

IBM i

Journal Code	Type	Description
J	PR NR RS	Previous receiver Next receiver Receiver saved
F	JM MS SS OP CL PT	Journaling started Member Save-while-active group saved File open File closed Recorded added
U	UB UP DL xx	Record updated before image Record updated after image Record deleted User entry

© Copyright IBM Corporation 2011

Figure 5-4. Journal codes

AS106.0

## Notes:

The Display Journal (**DSPJRN**) command is used to examine the file activity at a point in time. Listed above are examples of the types of journal entries a receiver may have accumulated as a result of activity against a file being journaled.

**DSPJRN** shows the activity against a journaled file in the form of journal entries. This short list of journal entry types allows you to more easily introduce the additional entries one gets when commitment control is used. We see those entries later in this topic as we look at more details of commitment control.

## Functions of commitment control

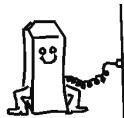
IBM i



- Allows transaction to be defined



- Insures all changes within transaction completed for all files affected if processing is interrupted



- Provides automatic backout of incomplete transaction



- Permits backout by user



- Provides for restart of an application in the event of job or system failure

© Copyright IBM Corporation 2011

Figure 5-5. Functions of commitment control

AS106.0

### Notes:

*Commitment control* is a function that allows you to define and process a group of changes to the database files or tables as a logical unit of work.

A *logical unit of work (LUW)* is defined as a group of individual changes to objects on the system that should appear as a single atomic change to the user. End users and application programmers call this a *transaction*.

Commitment control ensures that either the entire group of individual changes occur on all systems that participate, or that none of the changes occur.

An example of changes that must be processed together (or not at all) is the transfer of funds from a person's savings account to the same person's checking account. To the user, this is a single transaction. However, more than one change occurs to the database because both savings and checking accounts are updated.

Commitment control can be used to design an application so that it can be restarted if a job, an activation group within a job, or the system ends abnormally.

The application can be started again with assurance that no partial updates are in the database due to incomplete logical units of work from a prior failure.

Commitment control helps you with recovery and restart of RPG IV applications in the event of a system or job failure. Journaling without commitment control would force you to follow these steps in order to recover or restart:

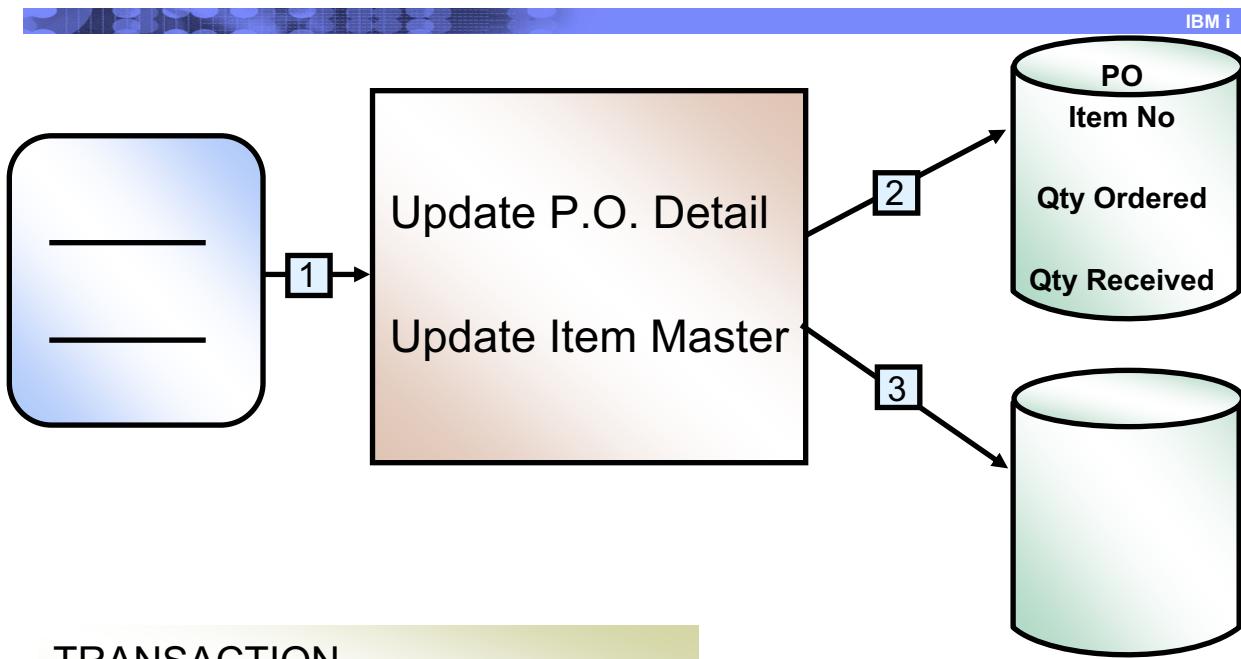
1. Review the history log to locate active jobs
2. Determine the files in use that are using **DSPJRN**
3. Identify each transaction within each job
4. Determine if transaction is complete or not
5. Check for multiple jobs using the same record
6. Allocate the files exclusively during the recovery
7. Enter adjustment transactions as necessary

The more complex your transactions, the more difficult the recovery becomes.

Commitment control supplies additional file recovery capability to the journaling function, as follows:

- Allows definition of complex transactions within an application
- Ensures that the complete transaction is processed
- Provides automatic backout of incomplete transactions
- Provides automated user backout of transactions
- Provides sufficient information for a job restart

# Transaction definition



## TRANSACTION

**A group of changes to database files that appear as a single change to the user.**

© Copyright IBM Corporation 2011

Figure 5-6. Transaction definition

AS106.0

## Notes:

Most jobs, except for file maintenance jobs, process database records in groups. An application transaction often affects several records. Groups can include multiple records from the same file or from multiple files.

Commitment control allows the programmer to define groups of transactions as one whole unit. Commitment control allows programmers to define logical transaction boundaries such that the entire group of changes occur or none of them occur at all.

A *transaction* is defined as a group of changes that are made to database files that appear to be a single change to the workstation user. This is also called a *logical unit of work*. For example, a user enters information to log the receipts against an open purchase order. Complete processing of this transaction includes having the program update both the purchase order detail file and the item master file.

Commitment control requires that certain programming be performed and is easiest to incorporate as part of initial application design. It can be difficult to add commitment control to an established application because a good knowledge of the data and what constitutes a transaction is required.

# Establishing a commitment control definition

IBM i

1. Identify files updated by complex transactions
2. Begin journaling the physical file (PF) accessed under commitment control
3. In job:
  - Start commitment control before opening files
  - Monitor for program termination
  - End commitment control when it is no longer needed
4. In RPG IV program:
  - Specify files to be opened under commitment control
  - Define transaction boundaries

© Copyright IBM Corporation 2011

Figure 5-7. Establishing a commitment control definition

AS106.0

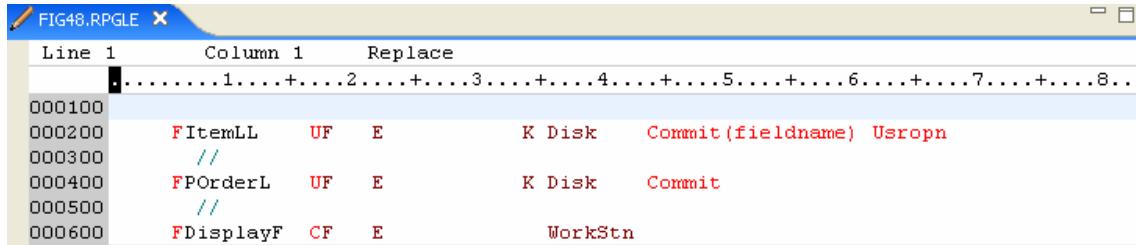
## Notes:

To establish the commitment control definition, you have to take these steps to set up the application:

1. Identify the files within an application that will be updated under commitment control
2. Ensure that the physical files being changed by your application are journaled to the same journal
3. Use the COMMIT keyword on the identified files in the F specifications (File specifications)
4. Determine the logical points within your application for the COMMIT and ROLBK operation codes
5. Place the job in the commitment control environment with **STRCMTCTL**
6. Process your application and stop the commitment control environment with **ENDCMTCTL**

# Specify files for commitment control open

IBM i



The screenshot shows an IBM i RPGLE editor window titled "FIG48.RPGL". The code is as follows:

```

Line 1      Column 1      Replace
.....1.....+.....2.....+.....3.....+.....4.....+.....5.....+.....6.....+.....7.....+.....8...
000100      FItemLL    UF   E           K Disk   Commit(fieldname) UsrOpn
000200      //                                 
000300      FPOrderL   UF   E           K Disk   Commit
000400      //                                 
000500      FDisplayF  CF   E           WorkStn
000600

```

© Copyright IBM Corporation 2011

Figure 5-8. Specify files for commitment control open

AS106.0

## Notes:

Use the keyword COMMIT to indicate that a DISK file is to be opened under commitment control. Only database files that are a part of your defined transaction are identified in this way. The program may have other files (for example, display files, or non-commitment control database files) that are not opened for commitment control.

The COMMIT keyword has an optional parameter (fieldname above that references the optional name field specified with the commit keyword) that allows you to specify conditional commitment control. If you use it, the RPG IV compiler creates a one-byte character field with the same name as was specified in the parameter. It is initialized with a value of 0 or off. If the field is set to 1 prior to the file being opened, the file is opened and runs under commitment control.

To set the COMMIT keyword parameter prior to opening the file, you can pass a value of 1 when the program is called. You can also explicitly set the 1 value in the calculation specifications before doing your own explicit file open with the OPEN opcode. Remember to use the USROPN file keyword in order to explicitly open your file in calculations if you prefer the explicit technique for conditional commitment control.

## Commitment control considerations

IBM i

- Prerequisite is journaling-related files to the same journal
- Must issue **STRCMTCTL** within each job using the function
- Start commit journal entry **SC** is issued by IBM i
- Commit journal entry is caused by program opcode
- Program rolls back uncommitted changes automatically
- Program can allow users to rollback changes
- Accessed records are locked until commit/roll back
- After failure, less analysis required for determining restart/recovery points
- Control language programs and RPG IV programs must be modified

© Copyright IBM Corporation 2011

Figure 5-9. Commitment control considerations

AS106.0

### Notes:

As mentioned previously, a prerequisite for commitment control is journaling. The file must be journaled before it can be opened by the program for commit. Each job using the program must issue a begin commitment control command to put the job in a commitment control environment. The start commit journal entry **SC** is entered by IBM i when I/O is done to a file opened for commit. The commit journal entry is caused by the program running the commit operation.

If uncommitted changes for a file exist and the job ends, or the end of a commitment control command is run, the uncommitted changes are rolled back. Uncommitted changes can also be rolled back under control through the program in which the ROLLBACK operation is coded. Uncommitted changes are rolled back at various times, depending on the way a job is ended:

- If the system terminates, rollback occurs at the next IPL.
- If the job fails abnormally or no ENDCMTCTL is specified, changes are rolled back at the next STRCMTCTL.
- If the job ends normally, changes are rolled back at the ENDCMTCTL.

# Journal receiver entries

IBM i

Journal Code	Type	Description
C	BC	Begin commitment environment
C	EC	End commitment environment
C	SC	Commit cycle started
C	CM	Set of changes committed
C	RB	Set of changes rolled back
R	UB	Record updated before image
R	UP	Record updated after image
R	BR	Before image rolled back
R	DR	Record deleted for rollback
R	UR	After image rolled back

© Copyright IBM Corporation 2011

Figure 5-10. Journal receiver entries

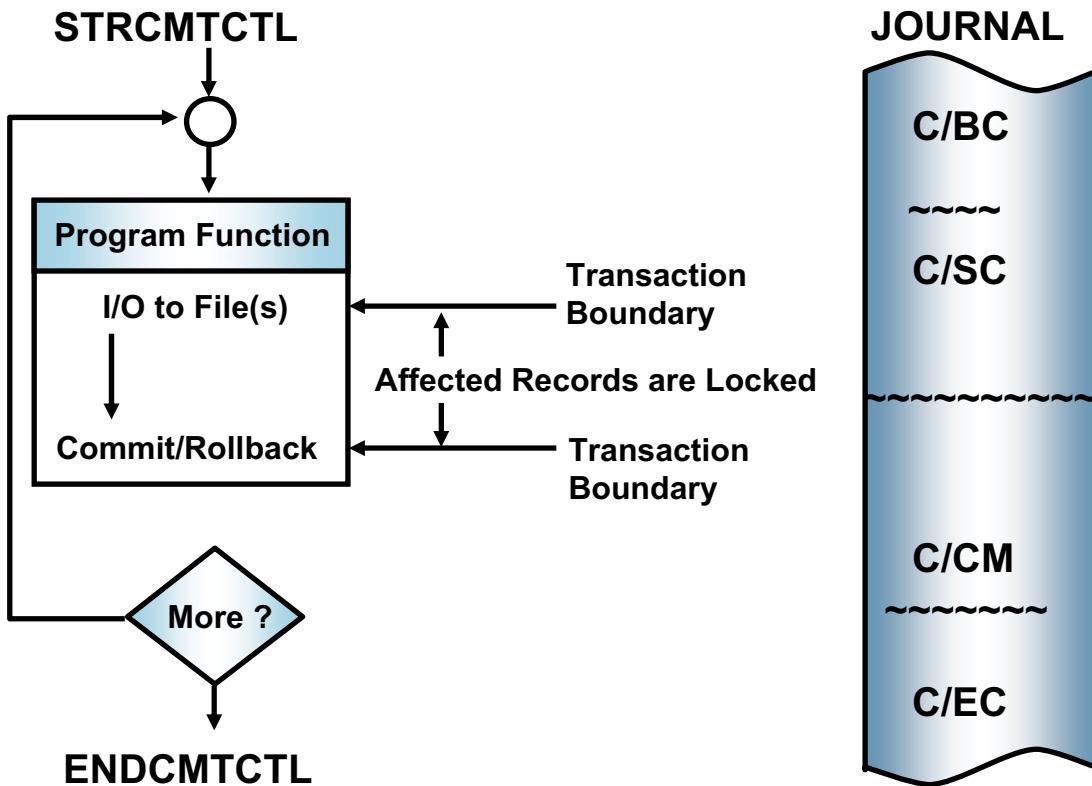
AS106.0

## Notes:

Commitment control adds specific entries to a journal receiver.

# COMMIT: All or nothing

IBM i



© Copyright IBM Corporation 2011

Figure 5-11. COMMIT: All or nothing

AS106.0

## Notes:

Typically done in the form of a CL program, the commitment control enabling sequence is:

1. The **STRCMTCTL** CL command places the job in a commitment control environment.
2. The **CALL** command calls the program that has specified the commitment control function.
3. The program opens the files under commitment control (program may also have database files that are not under commitment control). At this point a Start Commit (C/SC) entry is written in the journal for the file.
4. The program processes a transaction. This transaction is logged to the journal as well as subsequent transactions.
5. The program commits one or more transactions, or, if an error occurs, rolls back the transactions. If no error occurs and the transactions are successfully committed, a C/CM entry is written to the journal.
6. The **ENDCMTCTL** command ends the commitment control definition for the job. A C/EC entry is posted to the journal.

# Commit/rollback

IBM i

- COMMIT
  - 'ALL'
  - End transaction
  - Unlock records
  - COMMIT operation code
- ROLLBACK
  - 'NOTHING'
  - Reverse changes to all files back to last boundary
  - End transaction
  - Unlock records
  - ROLBK operation code

© Copyright IBM Corporation 2011

Figure 5-12. Commit/rollback

AS106.0

## Notes:

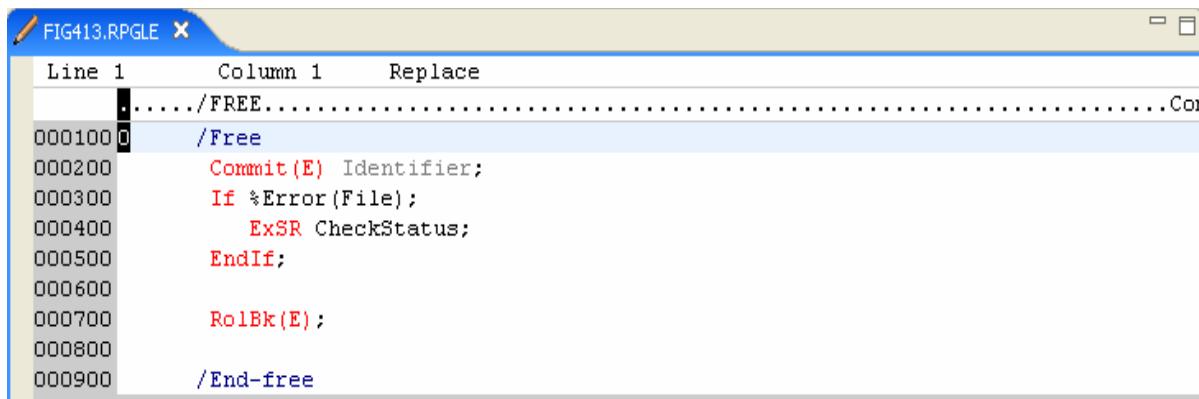
The COMMIT operation makes all the changes to the files opened under commitment control that have been specified in output operations since file opening, or the last commit or rollback operation.

The ROLBK operation eliminates or reverses all of the changes to the files that have been specified in output operations since the previous commit or rollback operation.

The rollback function occurs if there are uncommitted database changes and the user issues a program controlled rollback operation. It also occurs if the system issues the rollback due to job end, commitment control end, or system termination. Ending the program with uncommitted changes does not cause the rollback operation to occur until End of Job or ENDCMTCTL occurs.

# Syntax of COMMIT/ROLBK operations

IBM i



The screenshot shows an IBM i RPGLE editor window titled "FIG413.RPGLE". The code is as follows:

```

Line 1      Column 1    Replace
...../FREE.....Cor
000100 0   /Free
000200      Commit(E) Identifier;
000300      If %Error(File);
000400          ExSR CheckStatus;
000500      EndIf;
000600
000700      ROLBK(E);
000800
000900      /End-free

```

© Copyright IBM Corporation 2011

Figure 5-13. Syntax of COMMIT/ROLBK operations

AS106.0

## Notes:

The following are the valid RPG IV operation codes used with commitment control.

- COMMIT = All
  - Ends transactions
  - Unlocks records
- COMMIT IDENTIFIER
  - Source of restart information
  - Contents are controlled by program
  - Written to Entry Specific Data of associated journal entry
  - Written to notify object on automated rollback
- ROLBK = Nothing
  - Reverses changes to all files back to last boundary
  - Ends transactions

- Unlocks records

Implicit rollback occurs if uncommitted transactions and:

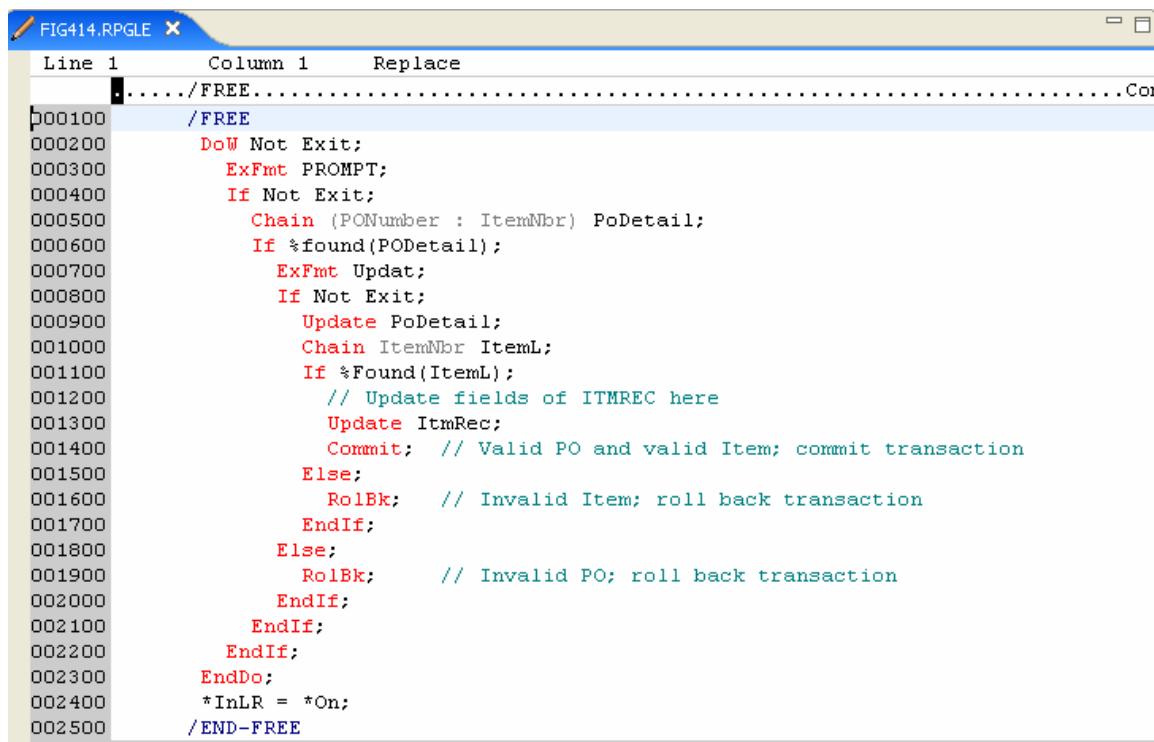
- **ENDCMTCTL**
- Normal Return
- **ENDJOB \*CNTRLD**
- ENDSBS \*CNTRLD
- IPL after abnormal system end

The E-extender can be used with COMMIT and ROLBK to test for an error condition. The %Status BIF can test status codes 802 through 805 for problems, as follows:

- 00802: Commitment control not active.
- 00803: Rollback operation failed.
- 00804: Error occurred on COMMIT operation
- 00805: Error occurred on ROLBK operation

# Calculations for commitment control

IBM i



```

FIG414.RPGLE X
Line 1      Column 1      Replace
...../FREE.....Cor
D00100      /FREE
000200      DoW Not Exit;
000300          ExFmt PROMPT;
000400          If Not Exit;
000500              Chain (PONumber : ItemNbr) PoDetail;
000600              If %found(PoDetail);
000700                  ExFmt Updat;
000800                  If Not Exit;
000900                      Update PoDetail;
001000                      Chain ItemNbr ItemL;
001100                      If %Found(ItemL);
001200                          // Update fields of ITMREC here
001300                          Update ItmRec;
001400                          Commit; // Valid PO and valid Item; commit transaction
001500                      Else;
001600                          ROLBK; // Invalid Item; roll back transaction
001700                      EndIf;
001800                  Else;
001900                      ROLBK; // Invalid PO; roll back transaction
002000                  EndIf;
002100                  EndIf;
002200              EndIf;
002300          EndDo;
002400          *InLR = *On;
002500      /END-FREE

```

© Copyright IBM Corporation 2011

Figure 5-14. Calculations for commitment control

AS106.0

## Notes:

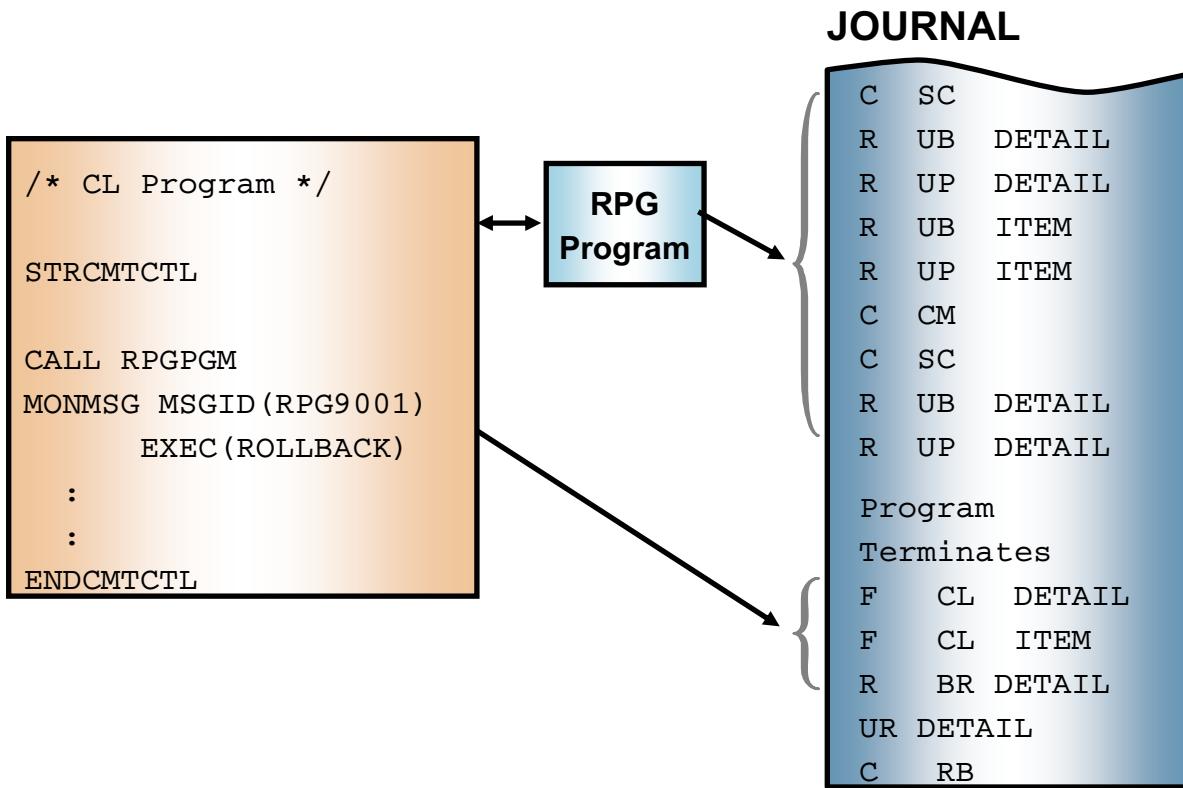
COMMIT defines transaction boundary.

ROLBK resets partial transaction when end of job is requested. A commitment cycle begins when a CHAIN operation uses PODETAIL. The record retrieved by the CHAIN is locked from update by other users as it is read. Even when the record is updated, it remains locked. The ITMREC record behaves the same way in that it remains locked, even after it is updated until the COMMIT operation is performed. Commit cycles should be kept as short as possible.

The program issues a rollback under two circumstances. If the ITMREC is not found at the end of the transaction entries for an item, the transaction is cancelled with a ROLBK. If in the middle of entering transaction records, the operator requests an exit from the program by pressing F3, ROLBK resets the partial transaction.

# End of program

IBM i



© Copyright IBM Corporation 2011

Figure 5-15. End of program

AS106.0

## Notes:

The CL program monitors for termination of the RPG IV program in case a rollback operation is necessary. The rollback would clean up any uncommitted transactions that are left from the terminated program.

## Automatic rollback occurs

IBM i

- When uncommitted changes
  - ENDCMTCTL + inquiry message
  - ENDJOB
  - Abnormal system end at next IPL

© Copyright IBM Corporation 2011

Figure 5-16. Automatic rollback occurs

AS106.0

### Notes:

## Notify object

IBM i

- Can obtain user-specified data (COMMIT ID) defined in program
- Can be used to restart applications
- Object types allowed:
  - \*FILE: Record added at end
  - \*MSGQ: Message CPI8399 added to queue
  - \*DTAARA: Entire area replaced (one per job or user)

© Copyright IBM Corporation 2011

Figure 5-17. Notify object

AS106.0

### Notes:

A *notify object* is:

- a database file,
- a message queue, or
- a data area

that contains information identifying the last successful transaction completed for a particular commitment definition if that commitment definition did not end normally. The information used to identify the last successful transaction for a commitment definition is given by the commit identification that associates a commit operation with a specific set of committable resource changes.

The commit identification of the last successful transaction for a commitment definition is placed in the notify object only if the commitment definition does not end normally. This information can be used to help determine where processing for an application ended so that the application can be restarted.

The notify object is a parameter of the **STRCMTCTL** command. It is used for logging information that can be helpful in restarting an application.

A database file is the most flexible type of notify object out of the three in terms of usability for recovery. It is easily accessed and can hold multiple entries (one per user). It can also be journaled for protection against loss, and it can be saved to a tape or SAVF. A data area can only contain information about one failure for a single job. The data is overwritten with each failure. A message queue can hold many messages, but the contents cannot be saved. This can make random access and retrieval awkward.

When a program is started after an abnormal end, the program can look for an entry in a notify object to assist the program to restart an LUW again. After the LUW has been started again, the notify object is cleared by the program to prevent it from starting the same LUW yet another time.

# How can a notify object be used?

IBM i

- DB file:
  - Query commit ID for restart information
- Message queue:
  - Send message to users regarding last transaction completed from user's MSGQ
  - Batch restart program can read message queue and retrieve restart data
- Data area:
  - Can write status information for batch application in DTAARA

© Copyright IBM Corporation 2011

Figure 5-18. How can a notify object be used?

AS106.0

## Notes:

Here are the ways that you can use a notify object:

- If the commit identification is placed in a database file, query this file to determine where to start each application or workstation job again.
- If the commit identification is placed in a database file that has a key or user name, the program can read this file when it is started. If a record exists in the file, start the program again. The program can send a message to the workstation user identifying the last transaction committed. Any recovery is performed by the program. If a record exists in the database file, the program deletes that record at the end of the program.
- If the commit identification is placed in a message queue for a particular workstation, a message can be sent to the work station users when they sign on to inform them of the last transaction committed.
- For a batch application, the commit identification can be sent to a message queue. A program that is run when the application is started can retrieve the messages from the queue and start the programs again.

- For a batch application, the commit identification can be placed in a data area that contains totals, switch settings, and other status information necessary to start the application again. When the application is started, it accesses the data area and verifies the values stored there. If the application ends normally, the data area is set up for the next run.

## When is a notify object updated?

IBM i

- Abnormal job end or abnormal system end after one successful commit
- Normal EOJ after at least one successful commit and uncommitted changes exist
- **ENDCMTCTL** with uncommitted changes, at least one successful commit, and:
  - Batch job
  - Interactive job sent error message CPA8350 and response = RB or CM

© Copyright IBM Corporation 2011

Figure 5-19. When is a notify object updated?

AS106.0

### Notes:

A notify object is updated automatically under the conditions shown in the visual. It is used as part of programmer-written error-handling procedures.

If the job is interactive and is sent the error message CPA8350 *ENDCMTCTL requested with changes pending (RB C CM)* and the response is either RB or CM, the notify object is updated. For the CM response, the commit identifier is entered on the prompt display.

# Restart logic

IBM i

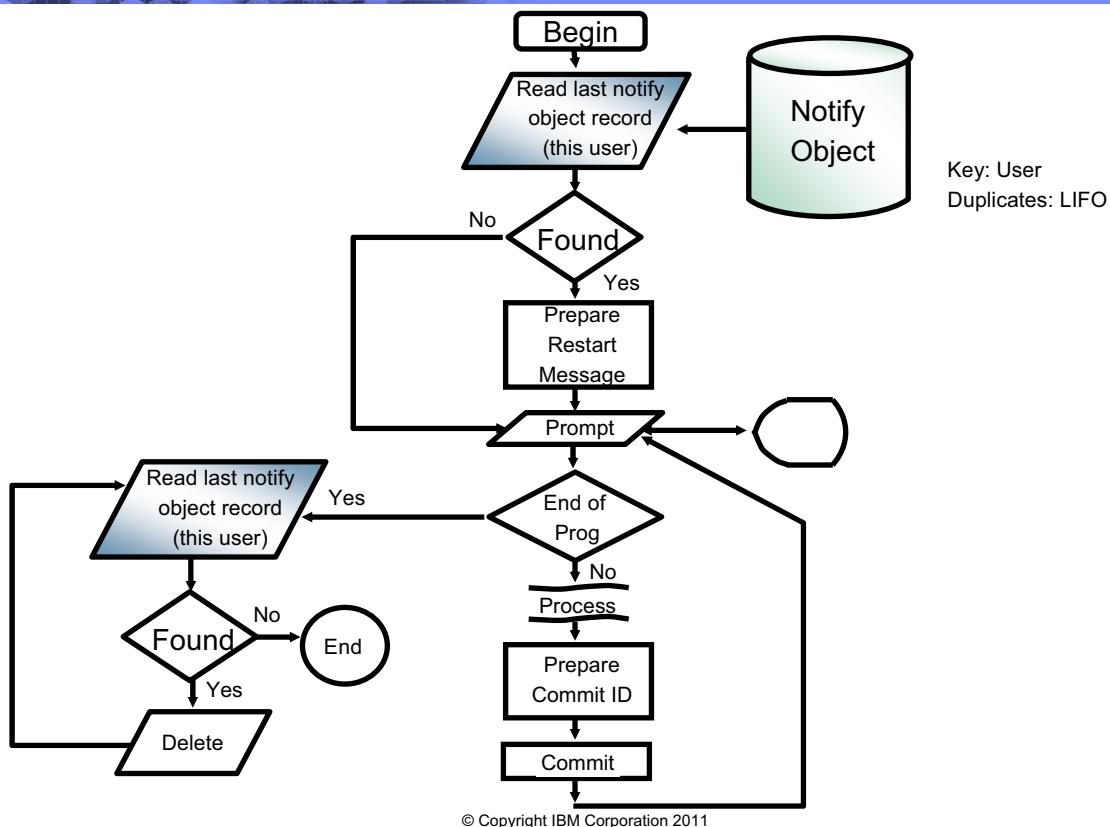


Figure 5-20. Restart logic

AS106.0

## Notes:

Adding restart logic to the program involves reading the notify object at the beginning of the program. There should be a status field in the record that indicates whether or not restart is necessary. If restart is needed, the program should use the information in the notify object's record to rebuild the application display up to that point. The user should be allowed to complete the transaction and then continue.

Note that the information recorded in the notify object is the commit identifier from the last successful commit cycle. In other words, rollback has occurred back to the previous commit boundary.

# Commitment control with notify object example (1 of 6)

```
*CCTLDEMOCL.PGM
Line 1          Column 1      Replace 1 change
+-----+-----+-----+-----+-----+-----+-----+-----+
002000 PGM
002100 DCL VAR(&REPLY) TYPE(*CHAR) LEN(1)
002200 DLTLIB LIB(CCTLLIB)
002300 MONMSG MSGID(CPF2110)
002400 CRTLIB LIB(CCTLLIB) TEXT('Temporary library for commitment -
002500 control demo')
002600 CPYF FROMFILE(*CURLIB/CUSMST) TOFILE(CCTLIB/CUSMST) -
002700 MROPT(*ADD) CRTFILE(*YES)
002800 CRTPF FILE(CCTLIB/CCTLNFYOBJ) SRCFILE(*CURLIB/QDDSSRC)
002900 CRTJRNRCV JRNRCV(CCTLIB/CCTLRCV)
003000 CRTJRN JRN(CCTLIB/CCTLJRN) JRNRCV(CCTLIB/CCTLRCV)
003100 CRTSAVF FILE(CCTLIB/CCTLSAVF)
003200 STRJRNPF FILE(CCTLIB/CUSMST) JRN(CCTLIB/CCTLJRN) IMAGES(*BOTH)
003300 SAVOBJ OBJ(CUSMST) LIB(CCTLIB) DEV(*SAVF) OBJTYPE(*FILE) -
003400 SAVF (CCTLIB/CCTLSAVF)
003500 CONTINUE: +
003600 STRCMCTL LCKLVL(*CHG) NFYOBJ(CCTLIB/CCTLNFYOBJ (*FILE)) -
003700 DFTJRN (CCTLIB/CCTLJRN)
003800 OVRDBF FILE(CUSMST) TOFILE(CCTLIB/CUSMST)
003900 CALL PGM(CCTLDEMO)
004000 ENDCMCTL
004100 RUNQRY QRY(*NONE) QRYFILE((CCTLIB/CCTLNFYOBJ)) OUTTYPE(*DISPLAY)
004200 OVRDBF FILE(CCTLNFYOBJ) TOFILE(CCTLIB/CCTLNFYOBJ)
004300 CALL PGM(CCNOTIFYR)
004400 DLTOVR FILE(CCTLNFYOBJ)
004500 DLTOVR FILE(CUSMST)
004600 SNDUSRMSG MSG('Do you wish to delete the demo in CCTLIB -
004700 (Y/N)') VALUES('Y' 'N') DFT('Y') TOMSGQ(*EXT) MSGRPY(&REPLY)
```

© Copyright IBM Corporation 2011

Figure 5-21. Commitment control with notify object example (1 of 6)

AS106.0

## Notes:

In order to use a notify object as a restart tool, you must have started journaling and commitment control, specifying the notify object to be used and its type. In our example, you can see that we specify the notify object (**\*FILE**, **CCTLNFYOBJ**), to be used.

# Commitment control with notify object example (2 of 6)

IBM i

Line 1	Column 1	Replace
000100	A	R CCTLNFYFMT
000200	A	CUSTNO 6A
000300	A	CUSTNAME 25A
000400	A	TEXT 25A
000500	A	USERID 10
000600	A	PGMNAME 10
000800	A	FILENAME 10
000900	A	LIBRNAME 10
001000	A	TRANSTIME Z
001100	A	K USERID

© Copyright IBM Corporation 2011

Figure 5-22. Commitment control with notify object example (2 of 6)

AS106.0

## Notes:

The notify object can be designed to hold any information that you decide would help you in a restart situation. In our example, we are logging these fields:

- CUSTNO: Customer number of record maintained
- CUSTNAME: Customer name
- TEXT: Text describing transaction
- USERID: ID of user who was performing maintenance
- PGMNAME: Name of program maintaining file (multiple programs might maintain a single file)
- FILENAME: Name of file being maintained
- LIBRNAME: Name of library where file is located
- TRANSTIME: Timestamp of transaction

With information such as what we have included above, we can access data in the notify object by user and assist that user to restart from the point where the application failed.

# Commitment control with notify object example (3 of 6)

IBM i

```

P00100 A REF (CUSMST)
000200 A INDARA
000300 A R PROMPT
000400 A CA03 (03)
000500 A 4 30'CUSTOMER MASTER UPDATE'
000600 A DSPATR(HI)
000700 A 6 22'(Demonstration of Commitment Contr-
000800 A ol)'
000900 A DSPATR(HI)
001000 A 4 3USER
001100 A 4 68SYSNAME
001200 A 5 3DATE
001300 A EDTCDE(Y)
001400 A 12 22'Enter Customer Number . . . :'
001500 A CUSNO R D I 12 52
001600 A 21 3'F3=Exit'
001700 A COLOR(BLU)
001800 A 19 3'Press Enter to continue'
001900 A COLOR(BLU)
002000 A R ADDREC
002100 A CA03 (03)
002200 A 4 30'CUSTOMER MASTER UPDATE'
002300 A DSPATR(HI)
002400 A 6 22'(Demonstration of Commitment Contr-
002500 A ol)'
002600 A DSPATR(HI)
002700 A 4 3USER
002800 A 4 68SYSNAME
002900 A 5 3DATE
003000 A EDTCDE(Y)

```

© Copyright IBM Corporation 2011

Figure 5-23. Commitment control with notify object example (3 of 6)

AS106.0

## Notes:

This visual and the one that follows show the display file used by the maintenance program.

# Commitment control with notify object example (4 of 6)

IBM i

```

005400 A          4 3USER
005500 A          4 68SYSNAME
005600 A          5 3DATE
005700 A          EDTCDE(Y)
005800 A          19 3'Press Enter to continue'
005900 A          COLOR(BLU)
006000 A          21 3'F3=Exit'
006100 A          COLOR(BLU)
006200 A          21 13'F23=Delete'
006300 A          COLOR(BLU)
006400 A          9 11'Customer Number . . . :'
006500 A          11 11'Name . . . . . :'
006600 A          CUSNO R D O 9 35
006700 A          FSTNAM R B 11 35
006800 A          SURNAM R B 11 47
006900 A          13 11'Address . . . . . :'
007000 A          ADDR1 R B 13 35
007100 A          ADDR2 R B 14 35
007200 A          ADDR3 R B 15 35
007300 A          ADDR4 R B 16 35
007400 A          PSTCDE R B 16 62
007500 A          R CONFIRM
007600 A          WINDOW(*DFT 7 40)
007700 A          WDWBORDER((*COLOR BLU) (*DSPATR RI)-
007800 A          (*CHAR ' ''))
007900 A          3 3'Define action . . . '
008000 A          ACTION 1 I 3 22VALUES('C' 'R' 'Q')
008100 A          3 27'C = Commit'
008200 A          4 27'R = Roll Back'
008300 A          5 27'Q = Quit'

```

© Copyright IBM Corporation 2011

Figure 5-24. Commitment control with notify object example (4 of 6)

AS106.0

## Notes:

# Commitment control with notify object example (5 of 6)

IBM i

```

000100      FCusMst    UF A E          K Disk      UsrOpn
000200      F                      Commit
000300      F                      InfDS (IOFBDS)
000400      FCctlDemoD CF   E          Workstn  IndDS (WkstnIND)
000500
000600 >>1  D Notify     E DS           ExtName (CCTLNFTYOBJ)
000700  D* CustNo            6A
000800  D* CustName          25A
000900  D* Text              25A
001000  D* UserID             10A
001100  D* PgmName            10A
001200  D* FileName           10A
001300  D* LibrName           10A
001400  D* Transtime          Z
001500
001600  D IOFBDS             DS
001700  D File                83   92
001800  D Library              93   102
001900
002000  D PSDS                SDS
002100  D Program               1    10
002200  D User                 254  263
002300
002400  D WkstnInd             DS
002500  D Exit                  1N  Overlay(WkStnInd:3)
002600  D Delete                 1N  Overlay(WkstnInd:23)
002700
002800 /FREE

```

© Copyright IBM Corporation 2011

Figure 5-25. Commitment control with notify object example (5 of 6)

AS106.0

## Notes:

Focus your attention on the definition of the notify object. Notice that:

At >>1, we describe the data structure of the notify object using an externally described reference to the notify PF (from the previous visual).

# Commitment control with notify object example (6 of 6)

IBM i

```

004600      CustName = Surname;
004700      Transtime = %Timestamp();
004800      UserID = User;
004900      PgmName = Program;
005000      FileName = File;
005100      LibrName = Library;
005200 >>2    Commit Notify;
005300
005400      When Action = 'R';
005500          Rblk;
005600
005700      When Action = 'Q';
005800          Leave;
005900      Ends1;
006000
006100      Exfmt Prompt;
006200      Enddo;
006300
006400      *InLR = *ON;
006500      Close CusMst;
006600      Return;
006700

006900      // Subroutines
007000      //-----
007100      Begr *Inzst;
007200          Open CusMst;
007300          Exfmt Prompt;
007400      Endr;
007500      //-----
007600      Begr Change;
007700          Select;
007800          When Exit;
007900              *InLR = *ON;
008000          Close CusMst;
008100          Return;
008200
008300      When Delete;
008400          Delete CusRec;
008500          Text = 'Record Deleted:';
008600
008700      Other;
008800          Update CusRec;
008900          Text = 'Record Updated:';
009000      Ends1;
009100
009200      Endr;
009300      //-----
009400      Begr Add;
009500          Select;
009600          When Exit;
009700              *InLR = *ON;
009800          Close CusMst;

009900      Return;
010000
010100      Other;
010200          Write CusRec;
010300          Text = 'Record Added:';
010400      Ends1;
010500
010600      Endr;
010700      //-----
010800 /END-FREE

```

© Copyright IBM Corporation 2011

Figure 5-26. Commitment control with notify object example (6 of 6)

AS106.0

## Notes:

This is the rest of the program.

At >>2 in the Commit operation, we specify the data structure that holds the record to be written to the notify object in the event of a failure. The notify object would contain information about the last record updated at the last Commit that was successful.

The only logic that is added to this file maintenance program is Commit, Rblk and the notify object.

# Using Notify Object Recovery

IBM i

RJSLANEY  
10/28/03

## NOTIFY OBJECT RECOVERY

S102NBRM

### (Demonstration of Commitment Control)

This information was retrieved from the Notify Object  
Do you want to recover (Y or N)? \_

Customer Number . . . . : 30  
Customer Name . . . . : WHEADON  
Transaction Type. . . . : Record Updated:  
User ID . . . . . . . . : RJSLANEY  
Transaction Time. . . . : 2003-10-28-12.15.13.195000

Press Enter to continue  
F3=Exit

© Copyright IBM Corporation 2011

Figure 5-27. Using Notify Object Recovery

AS106.0

## **Notes:**

This is the display that could be presented to a user who starts the maintenance application after having experienced some kind of a failure.

The user should be given the option to restart or not. In your installation, you might always restart as a default.

The program extracts the userid (PSDS) and then chains to the notify object for the customer master file using the userid.

This type of recovery could be easily added to the coding shown in the previous visuals.

## 6.2. Database triggers

## What is a trigger?

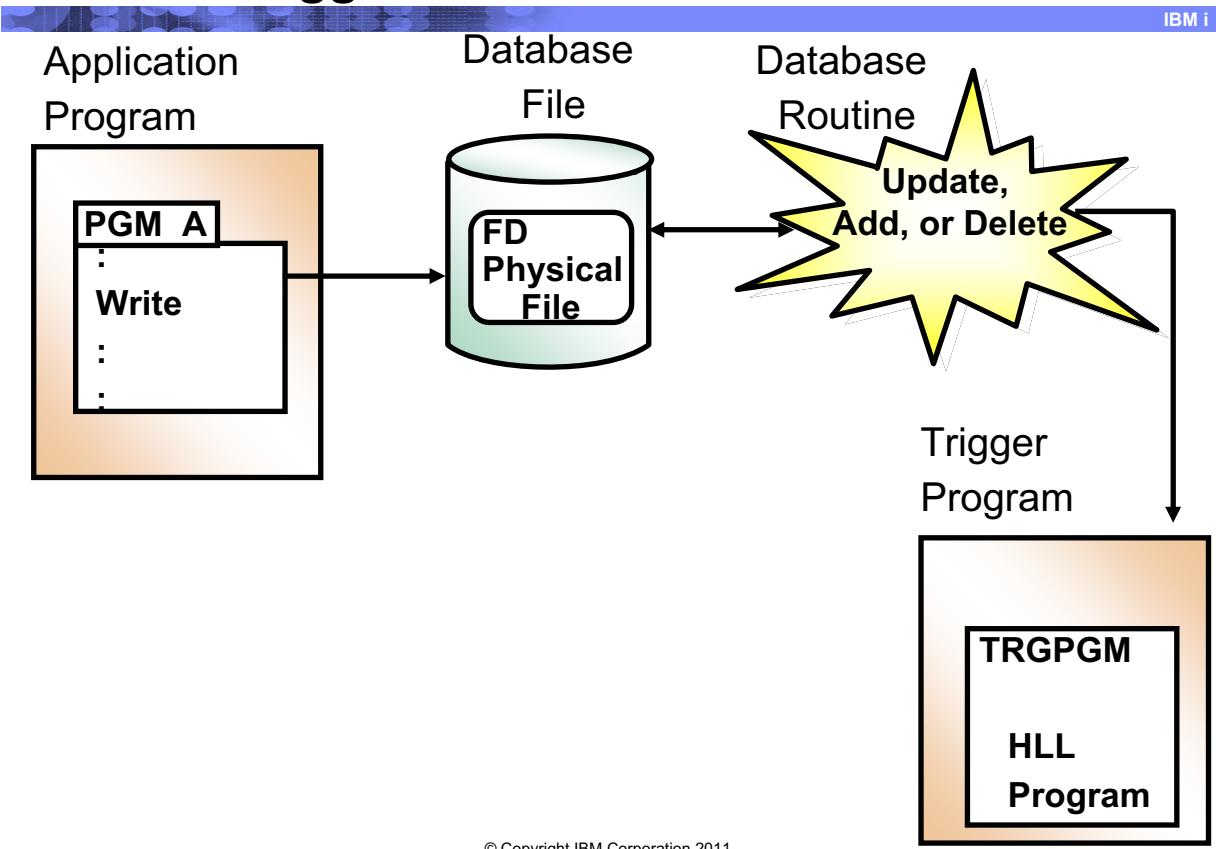


Figure 5-28. What is a trigger?

AS106.0

### Notes:

A trigger is a program that executes automatically when a change is made to a specified physical database file. The change to the database file can be an insert, update, or delete statement in an HLL program or any other program or utility.

Triggers can be used to:

- Enforce business rules
- Validate input data
- Write to other files for audit trail purposes
- Replicate data to different files to achieve data consistency

If your application environment needs to enforce particular business rules in the database, *DB2 UDB for i* offers the ability to automatically activate a user-written program called a trigger. A trigger performs any action that you specify. Triggers are run outside of applications that affect the contents of database. While a trigger runs, the application that fired the trigger waits for its completion. While you might think that the trigger was called by your program, what really happens is that a system database routine called by your

program fires the trigger. Triggers are associated with physical files and are activated no matter how the database change is made.

There are many situations where you can use triggers. Triggers can ensure that your database always complies with your business rules. They provide consistent checking and take the actions you code *every time* the data is changed. You can use them to monitor your critical files.

You can also use triggers to enhance existing applications and make them compliant with your specific business constraints. Triggers can sometimes provide a way to integrate different applications. For example, you might want changes made by an application to activate some procedure of another application.

Triggers can also play a major role when you want to develop client-server applications. You can use them to perform some application logic at the server side, depending on database changes. Applications at the client side do not need to take care of performing the functions that are performed by the triggers.

# Triggers: Introduction

IBM i

- Triggers are user-written programs
  - Change can be written in any IBM i HLL or in SQL
  - Associated with a physical file
  - Activated by the database before or after a database change
  - Independent from applications
  - Can be developed with any IBM i compiler
- When might you use triggers?
  - To consistently enforce complex business rules
    - Implement special application requirements
  - To monitor critical files
    - Validate data or security beyond object level
  - In a client-server environment
    - Each record added spawns processing tasks

© Copyright IBM Corporation 2011

Figure 5-29. Triggers: Introduction

AS106.0

## Notes:

Because triggers are written by i programmers, they can do almost anything any application program can do. They are not restricted to database functions or restricted to a particular language. They can even be written in SQL.

Why would you write a trigger rather than just call an application program? In some cases, there is a need to ensure that the function to be performed happens *every time* a specific database event occurs. For example, you might want to do something every time a record in the file is updated. The trigger fires (calls the program) regardless of what application performed the event.

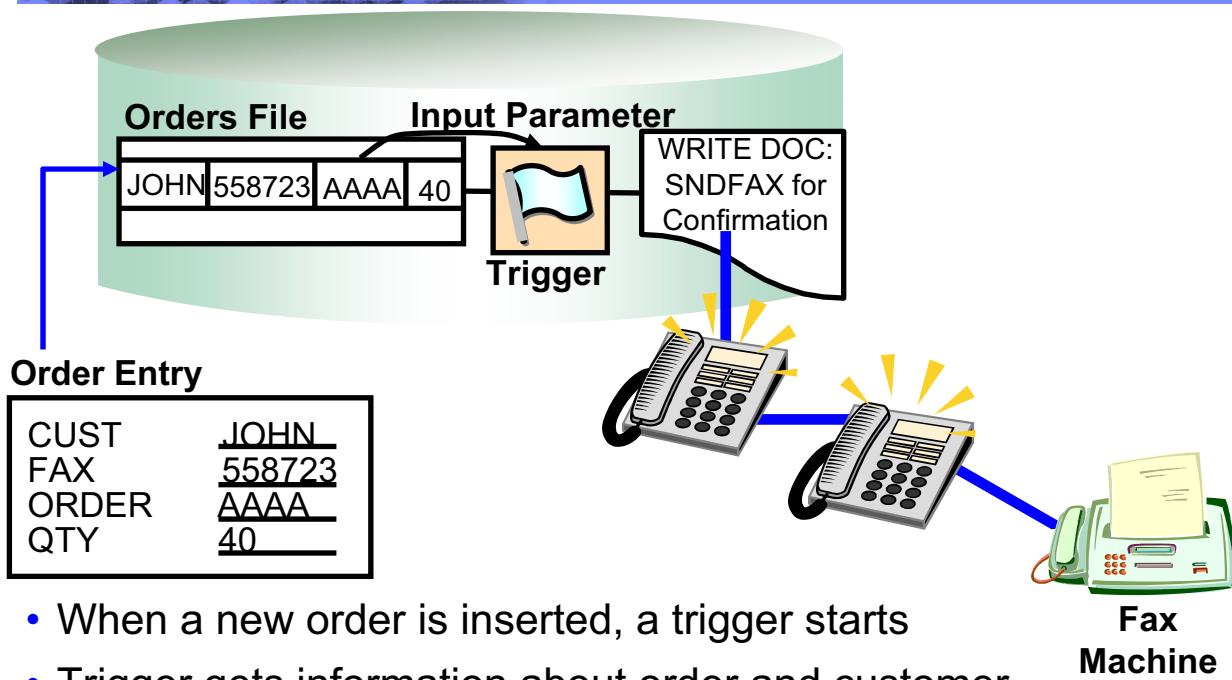
For example, the trigger could be made to fire if a record is updated by an application program, by the data file utility (DFU), by interactive SQL, or by a workstation application using the i as a server.

When your RPG IV (or other HLL) program processes an operation that changes the database file that has an attached trigger, the DB routine calls the trigger. Your program does not know that the trigger has been called. The trigger is independent of your program.

If a business policy for the database file changes, you would only code the change in the trigger, rather than all programs that access the DB file.

# Triggers: An example

IBM i



- When a new order is inserted, a trigger starts
- Trigger gets information about order and customer
- A confirmation fax is automatically sent

© Copyright IBM Corporation 2011

Figure 5-30. Triggers: An example

AS106.0

## Notes:

The visual shows an example of how triggers can be used to integrate a traditional application, such as order entry, with current technology. This could be the very common case of a company taking orders over the telephone, validating them and sending a confirmation fax to their customer afterwards. A trigger can be used to send a fax automatically to the customer as soon as the order is inserted into the database file. This can be done without changing the order-entry application. The trigger is called every time an order enters the orders file. The new record is passed automatically to the trigger program as an input parameter. The program might produce a predefined document with the new order and the new customer data and then send a fax to the customer.

It is important to point out the extraordinary flexibility that trigger programs offer. Other relational database platforms, for instance, allow triggers to be written using only proprietary languages. In some cases, triggers must have embedded SQL statements, and their functions are limited to database activity. On the i, triggers can be written in any supported compiled language and SQL. There is virtually no limit to the functions that can be performed in a trigger program. Triggers can take full advantage of any product, service, or feature that is supported on the IBM i system.

# Trigger components

IBM i

- Base file
  - The physical file controlling the trigger
- Trigger time
  - Timing in relation to the trigger event
  - Two possibilities:
    - Before or after the trigger event
- Trigger event
  - What causes the trigger to fire
  - Four possibilities:
    - Read, insert, delete, or update
- Trigger program
  - Your code

© Copyright IBM Corporation 2011

Figure 5-31. Trigger components

AS106.0

## Notes:

Here we see each individual component of the trigger environment.

# Adding a trigger to a file

IBM i

Add Physical File Trigger (ADDPFTRG)

Type choices, press Enter.

Physical file . . . . .	<u>Item_pf</u>	Name
Library . . . . .	<u>as10v6lib</u>	Name, *LIBL, *CURLIB
Trigger time . . . . .	<u>*After</u>	*BEFORE, *AFTER
Trigger event . . . . .	<u>*insert</u>	*INSERT, *DELETE, *UPDATE...
Program . . . . .	<u>mytrig</u>	Name
Library . . . . .	<u>as10v6lib</u>	Name, *LIBL, *CURLIB
Replace trigger . . . . .	<u>*NO</u>	*NO, *YES
Trigger . . . . .	<u>*GEN</u>	
<hr/>		
Trigger library . . . . .	<u>*FILE</u>	Name, *FILE, *CURLIB
Allow Repeated Change . . . . .	<u>*NO</u>	*NO, *YES

More...

F3=Exit F4=Prompt F5=Refresh F12=Cancel F13=How to use this display  
F24=More keys

© Copyright IBM Corporation 2011

Figure 5-32. Adding a trigger to a file

AS106.0

## Notes:

You can see the interaction of the trigger components in the **ADDPFTRG** command.

The **Add Physical File Trigger (ADDPFTRG)** command adds a trigger to call a named trigger program to a specified physical file. A *trigger* is a program instruction in a physical file that calls a trigger program when a specified operation is issued on the physical file. A trigger program is an exit program, called by a trigger, that contains a set of trigger statements.

The trigger program can be specified to be called before or after a change operation occurs. The change operation can be an insert, update, or delete operation through any interface. Change operations do not include clearing, initializing, moving, applying journal changes, removing journal changes, or changing end of data operations.

A maximum of three hundred triggers can be added to one physical file in order to call a trigger program before and after each allowable change operation. The trigger program to be called can be the same for each trigger or it can be a different program for each trigger.

An exclusive-no-read lock is held on the physical file when adding a trigger to that file. All logical files built over the physical file are also held with the exclusive-no-read lock.

Once a trigger is added to the physical file, all members of that specified file are affected by the trigger. When a change operation occurs on a member of the specified file, the trigger program is called. The trigger program is also called when a change operation occurs by way of either a dependent logical file or a Structured Query Language (SQL) view that is built over the physical file.

If you specify **\*LIBL** for the library of the file, **\*LIBL** is resolved at the time the trigger is added.

### **Trigger time (TRGTIME)**

Specifies the time when the trigger program is called.

- **BEFORE**

The trigger program is called before the change operation on the specified physical file.

- **AFTER**

The trigger program is called after the change operation on the specified physical file.

### **Trigger event (TRGEVENT)**

Specifies the event (the change operation to the physical file) that calls the trigger program. Each physical file can have one insert, one delete, and one update event for each trigger time specified. Only one event can be specified for each command issued.

- **INSERT**

An insert operation calls the trigger program.

- **DELETE**

A delete operation calls the trigger program.

- **UPDATE**

An update operation calls the trigger program.

- **READ**

A read operation calls the trigger program

### **Program (PGM)**

Specifies the name of the program that is called when the specified event occurs on the physical file. The program must exist on the system and be of object type **\*PGM**.

### **Allow Repeated Change (ALWREPCHG)**

Specifies whether repeated changes to a record within a trigger are allowed.

The possible values are:

- **NO**

Repeated changes to a record within a trigger are not allowed.

- **YES**

Repeated changes to a record within a trigger are allowed.

### **Trigger update condition (TRGUPDCND)**

Specifies the condition under which an update event calls the trigger program.

**Note:** This parameter applies only when **\*UPDATE** is specified on the **TRGEVENT** parameter.

The possible values are:

- **ALWAYS**

The trigger program is called whenever a record is updated, whether or not a value changes.

- **CHANGE**

The trigger program is called only when a record is updated and a value is changed.

### **Trigger**

As you can have more than one trigger for a time and an event, you must supply a name in order to uniquely identify which trigger you want to use.

**Note:** Triggers can also be added to a physical file (or SQL table) using the CREATE TABLE or ALTER TABLE SQL statements.

## Other trigger commands

IBM i

- **CHGPFTRG:** Enable or disable (temporarily) an existing trigger (or all triggers) for a file
- **RMVPFTRG:** Remove a specific trigger or all triggers

© Copyright IBM Corporation 2011

Figure 5-33. Other trigger commands

AS106.0

### Notes:

The Change Physical File Trigger (**CHGPFTRG**) command changes the state of one or all triggers for a file. The triggers have been defined with either the SQL CREATE TRIGGER or the Add Physical File Trigger (**ADDPFTRG**) command.

The Remove Physical File Trigger (**RMVPFTRG**) command removes the triggers that call trigger programs from a specified physical file. The triggers to be removed can be specified by trigger events, trigger times or trigger name. A trigger program is a program that has been added to the specified physical file by the Add Physical File Trigger (**ADDPFTRG**) command (system trigger) or the SQL CREATE TRIGGER statement (SQL trigger).

## The base file

- Must be a physical file
  - However, all logicals based on this file also fire the trigger
- When saved, trigger information is saved as well
  - Trigger information is stored with the file
    - For example, trigger program name, trigger event, and trigger time
  - Trigger programs, however, are *not* saved with the file
- If trigger program is renamed, moved, or deleted, file description information is not updated
  - You must remove and add the trigger again
  - **ADDPFTRG FILE**(*filename*/**\*LIBL** or **\*CURLIB**)
  - Resolved immediately and recorded in the file's description

© Copyright IBM Corporation 2011

Figure 5-34. The base file

AS106.0

### Notes:

Note that while you can only add triggers to physical files, the logical files based on a physical file with triggers enabled also fire trigger events.

Since trigger program objects are not saved with the files, it is recommended to put trigger programs in the same library as the database files to which they are related. Since saves are typically done on a library level, triggers will be saved with the files.

Since the trigger library is resolved when the trigger is added to a physical file, you should be aware of the following implications:

- **Renaming, Moving and Recompiling a Trigger**

This operation can be done, since there is no *hard* link between the trigger and the database file. If you change, delete, or move a trigger to another library, the data change operation on the associated file always fails because the system is not able to locate the trigger program.

- **Saving and Restoring**

When you save a database file, the trigger information is saved in the object description, unless you save it with **TGTRL(\*PRV)**. The trigger program, however, has to be saved separately. You might create the trigger programs in the same library as the associated file, so that a **SAVLIB** command would save all the objects.

- **Create Duplicate Objects and Copy File**

When you use **CRTDUPOBJ** or the **CPYF** commands to create a copy of a database file in a different library, the trigger information is not changed. If you need to create a copy of both trigger programs and database files to a different library, consider using the command **CPYLIB** or **CRTDUPOBJ OBJ(\*ALL)**. In these cases, the system updates the trigger library information in the file description if the triggers and the database file are in the same library. The system does not update the file description if you duplicate the objects one by one.

You can also create a trigger using the SQL statement CREATE TRIGGER. System i Navigator can be used in place of many of the file maintenance commands and utilities.

## Trigger time: Before and after

- BEFORE triggers can stop the event
  - For example, a data validation routine discovers an error on an update event and needs to prevent the update
  - Before triggers can also change/update the record image
    - If Allow Repeated change, \*YES specified
- AFTER triggers may or may not stop the event
  - Unless using commitment control
  - Useful for follow-up activity, spawning background tasks
- Trigger program can be specified for any unique combination of trigger event and time
  - Up to 300 triggers can be specified for a file
    - Insert, update, read, and/or delete
    - BEFORE and/or AFTER

© Copyright IBM Corporation 2011

Figure 5-35. Trigger time: BEFORE and AFTER

AS106.0

### Notes:

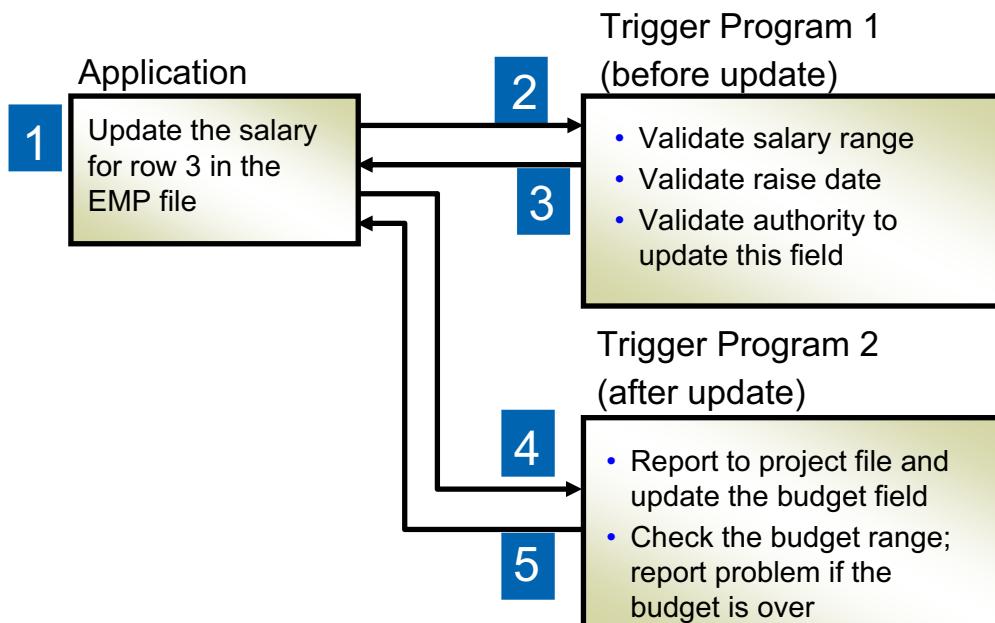
A file potentially could have up to 300 triggers: before and after insert, before and after update, and before and after delete.

You do not have to define triggers for all events and times. You might decide, for example, that an after-insert trigger is all you need. Likewise, you do not have to specify different trigger programs for different events and times; the same trigger program can be used for any or all events and times. For instance, you could assign the same trigger program to an after-insert and a before-delete.

A trigger program does not have to be unique to a file. The same program may be used for more than one file.

# Trigger time example: Triggers before and after a change operation

IBM i



Assumption: All files are opened under the same commitment definition

© Copyright IBM Corporation 2011

Figure 5-36. Trigger time example: Triggers before and after a change operation

AS106.0

## Notes:

In this visual:

1. The application tries to update the salary field for an employee in the EMP file.
2. The system calls the before update trigger before the record is updated. The before-update trigger validates all the rules.
3. If the validation fails, the trigger signals an exception informing the system that an error occurred in the trigger program. The system then informs the application that the update operation fails and also rolls back any changes made by the before trigger. In this situation, the after-update trigger is not called.
4. If all rules are validated successfully, the trigger program returns normally. The system then does the update operation. If the update operation succeeds, the system calls the after-update trigger to perform the post-update actions.
5. The after-update trigger performs all necessary actions and returns. If any error occurs in the after-update trigger program, the program signals an exception to the system.

The system informs the application that the update operation fails and all changes made by both triggers plus the update operation are rolled back.

## The trigger event

IBM i

- The trigger event causes the trigger to fire
  - Trigger event calls the trigger program
  - Trigger programs can be defined for any or all events
    - Same or different program can be used with different events
- The database event is at the record level
  - Any record-level update, insert, or delete can fire the trigger
  - File-level or field-level events are not supported
- A further qualification for update trigger events:
  - ALWAYS: Fire the trigger any time an application updates a record
  - CHANGE: Fire the trigger only if the record image on the update is changed from the before image

© Copyright IBM Corporation 2011

Figure 5-37. The trigger event

AS106.0

### Notes:

Note that trigger events have a *record-level scope*. This means that any time anything in the record is changed, an update trigger fires. You cannot specify that the trigger be fired only when a specific field's value changes, nor can you specify a trigger on an open or close event for a file.

## The trigger program

- Can be created by OPM or ILE HLL program
- Can be created by SQL trigger
  - Create trigger
  - System i Navigator
- May do almost anything any other program can do
- Passed a standard parameter block
  - First parameter contains information about record and changes
    - Typically coded in RPG as a data structure
    - Often called the *trigger buffer*
    - Contains a fixed-length part and a variable-length part
    - Contains the before and/or after images of the record
  - Second parameter contains the length of first parameter
- Application causing trigger to fire waits for trigger to complete

© Copyright IBM Corporation 2011

Figure 5-38. The trigger program

AS106.0

### Notes:

There are very few limits on what trigger programs can do. You could even write a trigger program to power down the system if you wanted to, but it is hard to imagine a situation where this would be a desirable feature.

Triggers can be created using a HLL or SQL. Our focus in this course is on triggers written in the RPG IV language.

Regardless of the language used, all trigger programs have two standard parameters passed to them by DB2 to identify which record is being processed.

The first parameter is a buffer consisting of two parts:

1. The first part contains a static amount of data about the file that triggered the program, the event, and more.
2. The second part contains the before and/or after images of the record being processed and null byte maps for each record.

The second parameter simply contains the length of the first parameter.

# Trigger buffer: Fixed portion of parameter

IBM i

Offset	Type	RPG Def	Description
0	CHAR(10)	10 A	Physical file name
10	CHAR(10)	10 A	Physical file library name
20	CHAR(10)	10 A	Physical file member name
30	CHAR(1)	1 A	Trigger event
31	CHAR(1)	1 A	Trigger time
32	CHAR(1)	1 A	Commit lock level
33	CHAR(3)	3 A	Reserved
36	BINARY(4)	10 I	CCSID of data
40	BINARY(4)	10 I	Current RRN
44	CHAR(4)	4 A	Reserved
48	BINARY(4)	10 I	Original record offset
52	BINARY(4)	10 I	Original record length
56	BINARY(4)	10 I	Original record null byte map offset
60	BINARY(4)	10 I	Original record null byte map length
64	BINARY(4)	10 I	New record offset
68	BINARY(4)	10 I	New record length
72	BINARY(4)	10 I	New record null byte map offset
76	BINARY(4)	10 I	New record null byte map length
80	CHAR(16)	16 A	Reserved

© Copyright IBM Corporation 2011

Figure 5-39. Trigger buffer: Fixed portion of parameter

AS106.0

## Notes:

The fixed portion of the trigger buffer contains several key elements:

- Physical name, library, and member identify the file that activated the trigger.
- Event has a value of **1** for insert, **2** for delete, **3** for update, or **4** for read.
- Time has a value of **1** for after or **2** for before.
- The commit lock level is important if the trigger can be used in a commitment control environment.
- The old record offset and old record length give the starting position and length of the old record image within the parameter.
  - The old record image is filled for update and delete operations.
- The new record offset and new record length give the starting position and length of the new record image within the parameter.
  - The new record image is filled for insert and update operations.

- The null byte maps are only applicable to triggers on record formats that have null-capable fields in them. DDS does not create null-capable fields by default. However, the default for SQL is null-capable. Each byte represents one field, **0** for Not NULL or **1** for NULL.

## Trigger buffer: Variable portion of parameter

IBM i

- Offsets and lengths are all variable
  - Stored in fields from the fixed portion
  - Vary by record format
  - Hard code it for a specific record format
  - Or use RPG IV pointers and built-in functions for a more flexible approach
    - And easier maintenance
  - Externally described data structures help too

Offset	Type	RPG Def	Description
*	CHAR(*)	* A	Original record image
*	CHAR(*)	* A	Original record null byte map
*	CHAR(*)	* A	New (updated) record image
*	CHAR(*)	* A	New record null byte map

© Copyright IBM Corporation 2011

Figure 5-40. Trigger buffer: Variable portion of parameter

AS106.0

### Notes:

# Coding example: Trigger parameter list

IBM i

001000	D TrigPlist	DS	
001100	D*		Trigger Buffer
001200	D FileName	10A	File Name
001300	D*		
001400	D LibName	10A	Library Name
001500	D*		
001600	D FileMember	10A	Member Name
001700	D*		
001800	D TrigEvent	1A	Trigger Event
001900	D*		
002000	D TrigTime	1A	Trigger Time
002100	D*		
002200	D CommitLockLvl	1A	Commit Lock Level
002300	D*		
002400	D Reserved1	3A	Reserved 1
002500	D*		
002600	D CCSID	10I 0	CCSID
002700	D*		
002800	D CurrentRRN	10I 0	Current Rrn
002900	D*		
003000	D Reserved2	4A	Reserved 2
003100	D*		
003200	D OldRecOffset	10I 0	Old Record Offset
003300	D*		
003400	D OldRecLen	10I 0	Old Record Len
003500	D*		
003600	D ORecNullOffset	10I 0	Old Rec Null Map Offset
003700	D*		
003800	D ORecNullLength	10I 0	Old Rec Null Map Length
003900	D*		
004000	D NewRecOffset	10I 0	New Record Offset
004100	D*		
004200	D NewRecLen	10I 0	New Record Len
004300	D*		
004400	D NRecNullOffset	10I 0	New Rec Null Map Offset
004500	D*		
004600	D NRecNullLength	10I 0	New Rec Null Map Length
004700	D*		
004701	D TrigBuffRes	16A	Reserved
004702	D*		
004703	D* Add record image fields here		

© Copyright IBM Corporation 2011

Figure 5-41. Coding example: Trigger parameter list

AS106.0

## Notes:

This visual shows an RPG DS that we have written based on the parameter list that DB2 UDB for IBM i automatically passes to triggers when they are called. You *must* define these parameters when you code trigger programs.

The static portion of the trigger buffer is supplied as a source member TRGBUF in QSYSINC/QRPGLESRC. You may copy this member into your QRPGLESRC source file and tailor it appropriately, as we have done in the above example, or use it as a guide to code your trigger parameter DS in your program. We show both methods.

Most fields have an obvious meaning. Here we describe how you can use the information that is automatically passed to trigger programs.

- **Trigger Event**

This information is particularly useful when the same trigger has been associated with different events. You might want to develop a common trigger for all kinds of I/O operations and take the appropriate actions depending on the operation types.

- **Commit Level**

Triggers are application-independent. Therefore, they can be activated either with or without commitment control, depending on the commit lock level of the underlying application.

This parameter provides a way to determine at run time whether the trigger is running under commitment control. It also determines the commit lock level. Database files should be opened with the appropriate commitment control option when they have one or more triggers associated with them. Dynamic commitment definition for files is provided by C and RPG IV. Triggers with embedded SQL statements can exploit the new SET TRANSACTION statement to dynamically adjust the commit lock level to the proper value.

- **Old and new record images**

Trigger programs are provided the images of the record before and after the database change. This information is essential when the trigger needs to perform data validation.

- **Old and new record null map**

This parameter is an array of as many characters as the number of fields in the database record. If the corresponding fields are null, the character will be set to 1.

The layouts for old and new records (and their related null maps) are unique to the file/program, so the layout of the first parameter depends upon the length of the record, and the length of the parameter cannot be generalized beyond position 96. The offset values should be used to determine where the record images are located in the buffer.

**Note:** The above only defines the fixed portion of the format. The data areas for the original record, null byte map of the original record, the new record, and the null byte map of the new record are of varying length and would immediately follow what is defined here.

# Coding example: Trigger program: Hard-coded buffer



```

>2  D ItemTrig      PI
D   TrigBuff           1000A Options(*Varsize)
D   Bufflength          10I 0 Const

>3  D/COPY AS10V1LIB/QRPGLESRC,TRIGPLIST
// Variable portion of trigger buffer
>4  D ORecord            52
D ONullMap             8
D NotUsed              20
D NRecord               52
D ITEMJLBMp             18A 0
D TimeChar              14A Overlay(Timestamp)

C                               Time                  Timestamp
/FREE
// Set values of parameters
>7  TrigPlst = TrigBuff;
Trigbufflen = Bufflength;
// Move the new record image into the work fields
>8  ItemRecord = ORecord;
//
>9  ItemRecord = NRecord;
Message = 'Update to ITEM_PF record ' + %char(ItmNbr)
+ ' at ' + TimeChar;
Dsply Message '*REQUESTER';
>10 Message = 'Record Length = '
+ %char(OldRecLen);
Dsply Message '*REQUESTER';

Message = 'Offset of original record = '
+ %char(OldRecOffset);
Dsply Message '*REQUESTER';
Message = 'Offset of New record = ' + ' ' + %char(NewRecOffset);
Dsply Message '*REQUESTER';
Return;
/End-Free

```

© Copyright IBM Corporation 2011

Figure 5-42. Coding example: Trigger program: Hard-coded buffer

AS106.0

## Notes:

This is an example of a trigger. Note that all key data elements are defined using external definitions.

This trigger is called by the database manager whenever an update to a record in the ITEM\_PF (physical file) is made:

1. We define the prototype for the trigger. Notice that the length of the buffer is variable as it includes a copy of the before (old) and after (new) record image. It will also contain an old and new null byte map for any null-capable fields. ITEM\_PF does not have any null-capable fields, so we do not have to worry about this. Also notice that as this is an OPM program, we use the EXTPGM keyword on the prototype (PR).
2. We define the procedure interface.
3. The data structure for the trigger buffer (fixed portion) is copied from a source member. This member is shown in the previous visual.
4. One way of accessing data in the old and new records is to define them as a part of the trigger buffer. Since they vary in length depending on the file with which you are

working, these fields are defined as a part of the program. Using pointer based variables is a better way to access the record data.

5. The second parameter to be received by our trigger program is defined. The length of the trigger buffer varies depending on the record length and the number of fields in the file with which we are working.
6. The format of the ITEM\_PF record format is copied into a data structure.
7. The parameters received are mapped to the program variables.
8. We move the new record map from the trigger buffer to the ITEM\_PF data structure defined in the D-specs.
9. A message that logs the transaction is sent to our message queue.
10. A second message is sent to the queue. This one uses the new record map.

# Coding sample: Trigger program: Better method

IBM i

```

000100 >>3 D Buffer      DS      1000  Based(StaticPtr)
000200  D TNAME
000300  D TLNAME
000400  D TMNAME
000500  D TEVENT
000600  D TETIME
000700  D TCMLCK
000800  D TFILLER1
000900  D TCCSID
001000  D TRRN
001100  D TFILLER2
001200  D TOLDOFF
001300  D TOLDLEN
001400  D TOLDNOFF
002500  D TPDMLRY
002100 // NOTE: Old & New record layouts (and null maps) not physically
002200 // coded in Buffer structure to retain flexibility
002300 >>4 D StaticPtr    S      *
002400 >>5 D NewPtr      S      *
002500
002600  D ItemTrig     PR      ExtPgm('ITMTRGBP')
002700  D
002800  D
002900
003000  D ItemTrig     PI
003100  D TrigBuff
003200  D BufLength
003300
003400 >>1 D OItemRec    E DS      ExtName(Item_PF) Prefix(Old_)
003500
003600  D NItemRec    E DS      ExtName(Item_PF) Prefix(New_)
003700 >>6 D
003800
003900 /Free
004000 >>7  StaticPtr = %Addr(TrigBuff);
004100 // Substring Method for decoding old record layout
004200 >>2  OItemRec = %Subst(TrigBuff : TOldOff+1 : TOldLen);
004300 // Pointer Method for decoding new record layout
004400 >>8  NewPtr = StaticPtr + TNNewOff;
004500 *InLR = *On;
004600 /End-Free

```

© Copyright IBM Corporation 2011

Figure 5-43. Coding sample: Trigger program: Better method

AS106.0

## Notes:

This example shows two methods to access old and new record data. One way is to use substring BIFs. The other is to use address pointers. Both methods avoid your having to code the layout of the old and new record (and null byte) images in the trigger buffer. Using either method makes your code less likely to be impacted when the buffer is changed because of a release update.

Using the substring method:

1. We define the old record map as an externally described data structure (as we did in the previous example)
2. We use the %Subst BIF to extract the old record image from the trigger buffer. The old record is in the buffer located at the **TOldOff+1**.

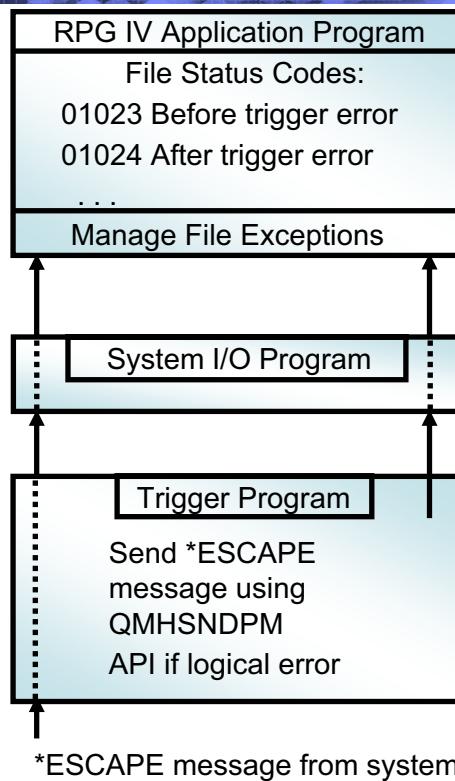
Using address pointers, we extract the new record image from the trigger buffer:

3. We base the address of the trigger buffer in the program using the **StaticPtr**.
4. We define **StaticPtr**.

5. We define **NewPtr**; this pointer points to the **NItemRec** DS which holds the new record image.
6. We base the address of the **NItemRec** DS on **NewPtr**.
7. We assign the address of the trigger buffer parameter (in the procedure interface (PI)) to **StaticPtr**. We now point to the first byte of the trigger buffer.
8. We add the offset of the new record image, **TNewOff** (from the beginning of the trigger buffer), to **StaticPrt** to get the address of the beginning of the new record image in the buffer. Since **NewItemRec** is based on **NewPtr**, we now have access to the data in the trigger buffer.

# Managing errors in triggers

IBM i



- Sequence of Events
1. RPG IV DB operation code
  2. Before trigger
  3. DB I/O performed
  4. After trigger

Figure 5-44. Managing errors in triggers

AS106.0

## Notes:

When you code error recovery for trigger programs, you have to be aware that triggers are *isolated* from your RPG IV application program because they are called by the database manager. Even the parameter list of a trigger is input-only.

If a failure occurs while the trigger program is running, an appropriate escape message must be signalled before the trigger ends. The message can be the escape message that is signalled by the system or a user-defined message that is retrieved from a message file by the trigger program.

There are two possible sources of errors:

1. System-generated error, such as a failure encountered accessing a locked record.

The system generates an exception and looks for an exception handler in the trigger. If none is found, the exception moves up the invocation stack in reverse order, searching for an appropriate exception handler. If the exception is not handled, the exception is processed by the system database module that is performing the I/O operation that fired the trigger. The I/O operation fails.

2. Failures that are detected by the trigger program.

This situation is very common in data validity checking. For example, assume that our insert trigger validates the records being inserted.

If the trigger determines that invalid data is being inserted, the insert operation must be rejected. You can send a user-defined message back to the application that fired the trigger or let the system database module handle the error.

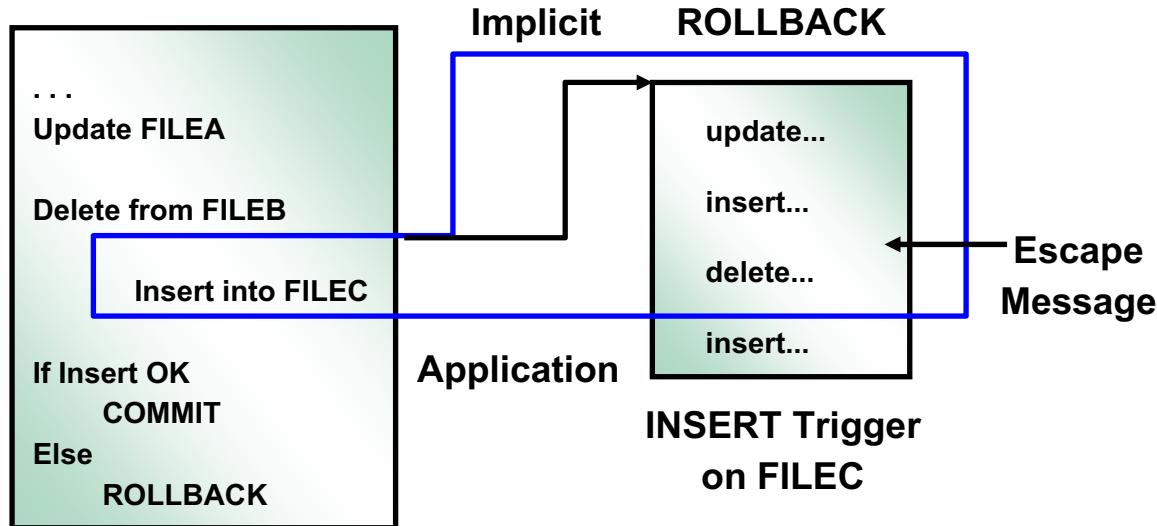
In both cases, the I/O operation fails, and the application receives an error message.

If no error message is signalled to the calling program after a trigger has failed, the database manager assumes that the trigger completed successfully and the operation that activated the trigger is completed as well. This is the case if there is a *logical error* in the trigger and it does not use QMHSNDPM API to send an \*ESCAPE message. If the trigger fails due to a system-generated exception, such as a lock timeout or a failed file open, the exception looks for an exception handler in the trigger. If none is found, the exception percolates back in search of an appropriate exception handler. The exception may reach the database module which is performing the I/O operation that fired the trigger. The I/O operation fails.

# Data integrity and trigger failures

IBM i

- Recommend application and trigger open files for commitment control
- Triggers and applications should run in same commitment definition



- In this scenario, no COMMIT or ROLLBACK allowed in triggers
- The application commits or rolls back the database changes
- Trigger changes and originating database change are rolled back

© Copyright IBM Corporation 2011

Figure 5-45. Data integrity and trigger failures

AS106.0

## Notes:

In order to ensure the best level of data consistency, we recommend that you use commitment control in your applications. There are implications in using commitment control with triggers for those resources that are accessed by the trigger programs. To avoid potential problems with data integrity, triggers and applications should share the same commitment definition. In this case, all the changes performed by triggers should be committed or rolled back by the application itself. The safest way to ensure that this happens is to compile your triggers with **ACTGRP(\*CALLER)**.

Triggers and applications should also share the same lock level.

If triggers run in a separate commitment control definition, they must commit or roll back their changes since the application is not able to do that. There are exposures of potential record locking and of consistency in this situation. If the trigger ends normally without committing its changes, the application is not able to release the locks on those records. Having different commitment definitions for triggers and applications should be pursued only if strictly necessary.

The visual illustrates what happens when a trigger ends abnormally due to a failure that is not monitored, and both trigger and application run under commitment control. The database change that fired the trigger is implicitly rolled back. This process does not affect other changes that were previously done by the application. The application can still choose to commit or roll back those changes.

## Updating the record

IBM i

- Before triggers may also modify the record firing the trigger
  - Only if Allow Repeated Update \*YES is specified
- Example:
  - Based on the value of a field that was set by the application program, the trigger program can change the values of other fields in the record before inserting or updating it in the file
    - You have a record containing a rate code and an exchange rate
      - If the exchange rate is zero, the trigger program could use the rate code to retrieve an exchange rate from the rates files and use it to set the exchange rate on the record before inserting or updating it
      - Or even just default the exchange rate to 1

© Copyright IBM Corporation 2011

Figure 5-46. Updating the record

AS106.0

### Notes:

Remember to specify the **ALWREPCHG(\*YES)** if you want the trigger to be able to update the record.

# Order of execution

IBM i

- Before-trigger
- Database change
  - \*RESTRICT
  - Non Referential Integrity (RI) errors, such as Record not found
- After-trigger
- Enforce all referential integrity delete CASCADE rules
- Enforce delete SET NULL/\*SETDFT
- \*NOACTION
- Enforce unique constraint

© Copyright IBM Corporation 2011

Figure 5-47. Order of execution

AS106.0

## Notes:

The following is a summary of the system's execution sequence of database and trigger events:

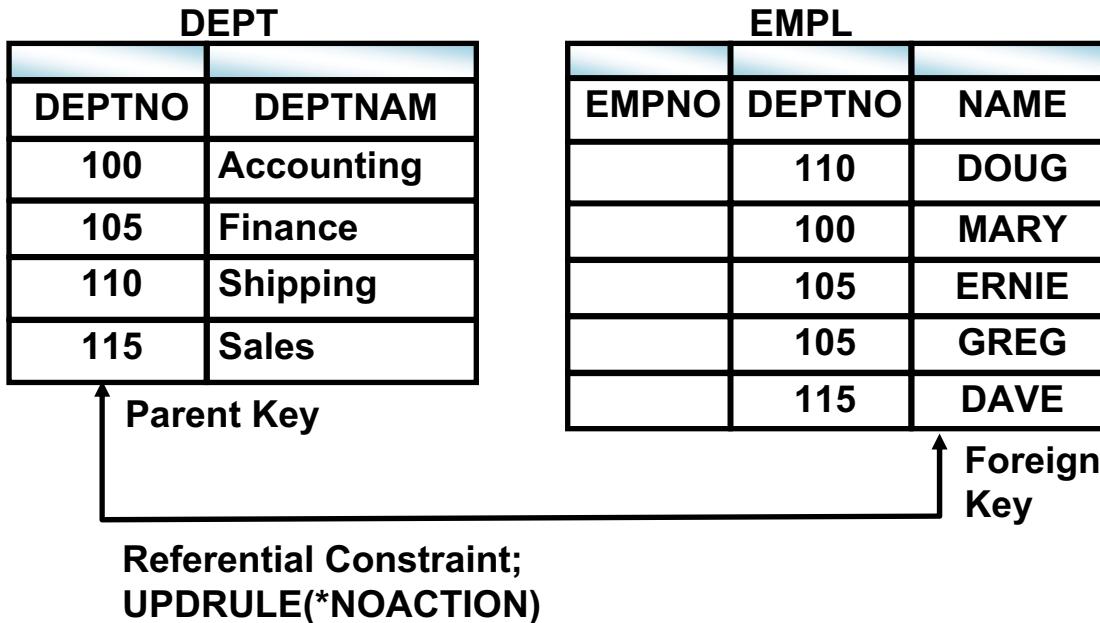
1. Run before-trigger statements, if there are any.
2. Run the change statement. If an RI RESTRICT rule (same as NO ACTION except that enforcement is immediate) is defined for this file, evaluate it now. Detect errors that are not associated with the RI constraints, for example, *Record not found*.
3. Run after-trigger statements, if there are any. Commit the enforcement cycle, and then begin enforcing the RI constraints.
4. Enforce all referential interrogate delete CASCADE rules (when a record in the primary file is deleted, delete all records with matching foreign keys).
5. Enforce delete SET NULL (set the value of all *nullable* foreign key fields to null when the corresponding primary key record is deleted).
6. SET DEFAULT (set all foreign key fields to their default when the corresponding primary key record is deleted).

7. Enforce NO ACTION rule.
8. Enforce unique constraint.

Any error during this execution cycle, such as an error in a trigger program or an RI constraint violation, stops the cycle. If there is an error, the system sends a message that identifies the error to the application.

## Trigger can perform update cascade

IBM i

**Desired:**

If DEPTNO is changed in DEPT file, update corresponding records in EMPL file.  
**ADDPFTRG FILE(DEPT) TRGTIME(\*AFTER) TRGEVENT(\*UPDATE) PGM(DEPTTRG)**

© Copyright IBM Corporation 2011

Figure 5-48. Trigger can perform update cascade

AS106.0

**Notes:**

Sequence of events:

1. A user changes the value of **DEPTNO** in the DEPT file from 105 to 125.
2. After-trigger DEPTTRG changes department number of ERNIE and GREG to 125.
3. Referential integrity (**\*NOACTION**) checked.

**No violation.**

## Machine exercise: Database triggers

IBM i



© Copyright IBM Corporation 2011

Figure 5-49. Machine exercise: Database triggers

AS106.0

### Notes:

Perform the machine exercise *database triggers*.

## 6.3. Embedded SQL: An overview

# Embedded SQL: Some questions

IBM i

Static?

What about maintenance?

Dynamic?

Can I change my code?

When are statements prepared?

Access plan?

Pass variables?



© Copyright IBM Corporation 2011

Figure 5-50. Embedded SQL: Some questions

AS106.0

## Notes:

There are different forms of embedded SQL. In this topic, we introduce embedded SQL but we do not discuss the subject in depth.

When you have choices, there are always considerations and questions.

In-depth training is available in two courses:

- *OL37/OL370 - Accessing the IBM i Database Using SQL*
- *OL38/OL380 - Developing System i Applications Using SQL*

# Coding choices

IBM i

- High-level language
  - READ/WRITE/CHAIN/UPDATE/DELETE
  - Embedded SQL in HLL programs
- **OPNQRYF**
  - Can be used to define set of records to be processed
  - Program processes subset
- Embedded SQL
  - Can be used to process SQL result sets
  - Can perform all I/O
  - Eliminates need for **OPNQRYF**

© Copyright IBM Corporation 2011

Figure 5-51. Coding choices

AS106.0

## Notes:

You have choices that you can make when you want to process one or more database tables and views in an HLL program. Usually, you are looking for flexibility and efficiency.

If you want to limit the rows processed or join several tables dynamically and do not want to keep the results permanently, you could choose to use **OPNQRYF** or embedded SQL.

A review of **OPNQRYF** follows:

The Open Query File (**OPNQRYF**) command enables a subset of records from a file to be selected for use during a particular execution of a program. It acts as a filter between the program and the database, so that the program only retrieves records that meet the criteria specified in the **OPNQRYF** command.

The command must be used in conjunction with an HLL program. The program may be run several times, without change, by simply changing the scope of **OPNQRYF** command on the database file.

**OPNQRYF** allows a dynamic definition of a view without using Data Description Specifications (DDS). No permanent object created, but a temporary file may be created.

This command is used to do any combination of the following database functions:

- Join records from more than one file, member, and record format. The join operation that is performed may be equal or non-equal in nature.
- Calculate new field values by using numeric and character operations on field values and constants.
- Group records by like values of one or more fields, and calculate aggregate functions, such as minimum field value and average field value, for each group.
- Select a subset of the available records. Selection can be done both before and after grouping the records.
- Arrange result records by the value of one or more key fields.

SQL can perform the same functions as **OPNQRYF** but from *within your program* rather than outside your program in CL. When you code embedded SQL, you are using the SQL language as an alternate means of performing I/O. However, SQL can be more powerful than native IBM i HLL I/O operations. Embedded SQL can provide the same power and efficiency as **OPNQRYF**, plus it offers you several unique features:

1. The result set can be composed within your HLL, and the data is processed only once. With **OPNQRYF**, the file is processed once by the **OPNQRYF** command and the subset is then processed by your program. This results in more overhead than using embedded SQL.
2. A group of rows can be processed by, for example, a single DELETE or UPDATE SQL statement.
3. An SQL statement can be constructed dynamically in your applications program. You pay a performance cost for this feature, but the cost can be offset by the flexibility that is offered by the program. Also, the cost to develop and maintain the program may be lower than using only static SQL.

## Why use embedded SQL?

IBM i

- SQL can be easier to read and maintain
- SQL can process multiple records in I/O operations
- Query Optimizer tunes SQL based I/O
- SQL can make code more portable

© Copyright IBM Corporation 2011

Figure 5-52. Why use embedded SQL?

AS106.0

### Notes:

Instead of using IBM i native database file operations, such as READ, CHAIN, UPDATE and DELETE, you can embed SQL statements in your RPG IV programs that process records in IBM i database tables.

SQL is the industry standard for database access and control and is used by programmers on many different platforms. Some reasons to consider using embedded SQL in RPG IV programs on IBM i system are:

- SQL can be easier to understand and maintain than the comparable RPG logic.

SQL I/O behaves more *logically* than RPG. Whereas in RPG, you must be careful not to issue an update before a read or a chain, SQL does not require a SELECT to be performed before an UPDATE.

In SQL, you always perform an operation on a table or a view. In RPG, you READ the filename but UPDATE the record format.

- SQL statements can perform an operation on a set of rows. A SELECT can produce a set of many rows.

SQL can simplify the program logic when multiple records are processed in an I/O operation, using UPDATE or DELETE.

- SQL operations are monitored and tuned by the query optimizer, which is enhanced regularly. Thus SQL I/O automatically takes advantages of the latest database enhancements that you have installed.
- Application migration *may* be simpler when using an industry standard language like SQL.

# What is embedded SQL?

IBM i

- SQL coded in HLLs
- Can perform same I/O operations as HLL
- Uses special source member type (SQLRPGLE)
- Preprocessor handles SQL
- Has two forms:
  - Static
    - SQL statements known at compile time
  - Dynamic
    - SQL statements created on-the-fly during execution

© Copyright IBM Corporation 2011

Figure 5-53. What Is embedded SQL?

AS106.0

## Notes:

Embedded SQL offers you many choices in developing your HLL programs. You may have already used SQL as a query tool by executing individual statements using System i Navigator (Run SQL Scripts), the interactive 5250 interface (STRSQL) and the 5250 tool, Query Manager.

You already know how powerful the SQL language is and now you want to know why you should and how you can use SQL in your programs rather than the I/O operations that are native to the HLL.

Let us review the nature of two forms of embedded SQL, static and dynamic.

- **Static SQL**

The source form of a static SQL statement is embedded within an application program written in a host language such as COBOL or RPG. The statement is *prepared* before the program is executed, and the operational form of the statement persists beyond the execution of the program.

A source program containing static SQL statements must be processed by an SQL precompiler before it is compiled. The precompiler checks the syntax of the SQL statements, turns them into host language comments, and generates host language statements to call the database manager.

The preparation of an SQL application program includes precompilation, the preparation of all static SQL statements, and compilation of the modified source program.

- **Dynamic SQL**

A dynamic SQL statement is prepared during the execution of an SQL application. The operational form of the statement persists until the last SQL statement leaves the call stack. The source form of the statement is a character string that is passed to the database manager by the program using the static SQL statement PREPARE or EXECUTE IMMEDIATE.

In summary, *static SQL* is any SQL statement that is fixed in form when the program is compiled. Static SQL can be and is *prepared* by the SQL precompiler for the HLL *before the program is compiled and executed*.

*Dynamic SQL* is any SQL statement that is not known and is created during execution of the program. These statements must still be prepared but are prepared during execution, **after preprocessing and compilation**.

# Inquiry program: Native I/O

IBM i

```

000002      FApInv_PF  IF   E          K Disk
000003      FEmbedOID  CF   E          Workstn InDDS(Indicators)
000004
000005      D  Indicators       DS
000006      D  Exit           1N  Overlay(Indicators:03)
000007      D  Cancel          1N  Overlay(Indicators:12)
000008      D  Error           1N  Overlay(Indicators:90)
000009
000011      /FREE
000012      Exfmt Prompt;
000013      Dow Not Exit;
000014
000015      Chain PoNbr ApInv_PF;
000016      Error = Not %Found(ApInv_PF);
000017
000018      Dow %Found(ApInv_PF);
000019      Exfmt Detail;
000020      Select;
000021      When Exit;
000022          *InLR = *ON;
000023      Return;
000024
000025      When Cancel;
000026      Leave;
000027      Ends1;
000028      Enddo;
000029
000030      Exfmt Prompt;
000031      Enddo;
000032
000033      *InLR = *ON;
000034      Return;
000035
000036      /END-FREE

```

© Copyright IBM Corporation 2011

Figure 5-54. Inquiry program: Native I/O

AS106.0

## Notes:

You recognize that this program gets a specific record from the APINV\_PF physical file and displays it. In the event of an error (not found condition), the user is prompted to re-enter a valid purchase order number.

For your reference, here is a display of a subset of the records in the APINV\_PF file:

04/24/01

## VENDOR LIST

OL88000

DIRTCITY

PO Number	Vendor Number	Vendor Invoice	PO Total			Date Due	Date Paid
			Amount	Net	Paid		
300071	10005	5000	1000.00	.00	04/11/1996	04/11/1996	
300072	10005	1005	2000.00	.00	06/30/1996	12/31/9999	
300073	10015	1580	3000.00	.00	04/11/1998	04/11/1998	
300074	10020	2007	4000.00	.00	05/25/1998	06/01/9999	
300075	10025	25697	500.00	.00	03/31/1998	12/31/9999	
300076	10030	301	600.00	.00	08/15/1998	12/31/9999	
300077	10035	350001	700.00	.00	03/01/1998	03/15/1998	
300078	10040	40079	800.00	.00	09/24/1998	12/31/9999	
300079	10045	45090	900.00	.00	06/25/1998	12/31/9999	
300080	10050	500777	100.00	.00	08/15/1998	12/31/9999	

More..

The following are the field definitions for the file:

Data Field	Field Type	Buffer Length	Buffer Length	Field Position	Usage
PONBR	PACKED	6	0	4	1
VNDNBR	PACKED	5	0	3	5
APINVNBR	CHAR	8	8	8	Both
APDATE	DATE	10	10	16	Both
POTOTAMT	PACKED	7	2	4	26
APDISCOUNT	PACKED	5	2	3	30
APNETPAID	PACKED	7	2	4	33
APSTATUS	CHAR	1	1	37	Both
APDATEPAID	DATE	10	10	38	Both
APCHECK#	PACKED	6	0	4	48
APDUEDATE	DATE	10	10	52	Both

And, here is the display file used in the program:

```

A                               REF(*LIBL/APINV_PF)
A                               INDARA
A                               CA03(03)

A          R PROMPT
A          3 30'Purchase Order Inquiry'
A          DSPATR(HI)
A          3 70DATE
A          EDTCDE(Y)
A          8 20'Enter Purchase Order Number:'
A          PONBR    R   D I 8 49
A 90                           ERRMSG('No such Purchase Order foun-
A                               d - rekey' 90)
A          21 8'Press Enter to Continue'
A          22 8'F3=Exit'
A          COLOR(BLU)

A          R DETAIL
A          CA12(12)
A          3 30'Purchase Order Inquiry'
A          DSPATR(HI)
A          3 70DATE
A          EDTCDE(Y)
A          8 20'Purchase Order:'
A          PONBR    R   O 8 47
A          10 20'Vendor Number:'
A          VNDNBR    R   O 10 47
A          11 20'Vendor Invoice Reference:'
A          APINVNBR  R   O 11 47
A          13 20'Cheque Number:'
A          APCHECK#   R   O 13 47
A          21 8'Press Enter to Continue'
A          22 8'F3=Exit F12=Cancel'
A          COLOR(BLU)

```

# Inquiry program: SQL I/O

IBM i

```

000001  FApInv_PF  IF   E      K Disk
000002  FEmbeddID  CF   E      Workstn InDDS(Indicators)
000003
000004  D Indicators    DS
000005  D Exit          1N  Overlay(Indicators:03)
000006  D Cancel         1N  Overlay(Indicators:12)
000007  D Error          1N  Overlay(Indicators:90)
000008
000009  /FREE
000010  Exfmt Prompt;
000011  Dow Not Exit;
000012  // SQL I/O Statements
000013
000014 >>  Exec SQL
000015 >>  Select VndNbr, ApinvNbr, ApCheck#
000016 >>  into :VndNbr, :ApInvNbr, :ApCheck#
000017 >>  from ApInv_PF
000018 >>  where PoNbr = :PoNbr;
000022
000023  // End SQL I/O Statements
000024  // Set Error indicator using SQLCOD
000025 >>  Error = (SqlCod <> 0);
000026
000027 >>  Dow SqlCod = 0;
000028  Exfmt Detail;
000029  Select;
000030  When Exit;
000031  *InLR = *ON;
000032  Return;
000033
000034  When Cancel;
000035  Leave;
000036  Ends1;
000037  Enddo;
000038
000039  Exfmt Prompt;
000040  Enddo;
000041
000042  *InLR = *ON;
000043  Return;
000044
000045  /END-FREE

```

© Copyright IBM Corporation 2011

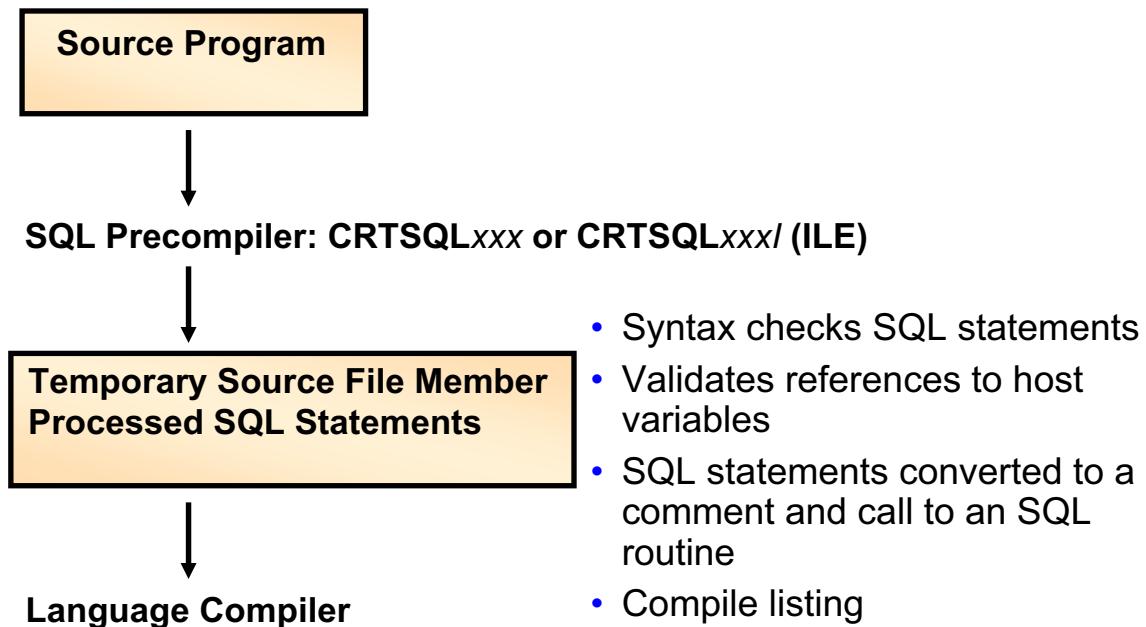
Figure 5-55. Inquiry program: SQL I/O

AS106.0

## Notes:

# Program generation

IBM i



© Copyright IBM Corporation 2011

---

Figure 5-56. Program generation

---

AS106.0

### **Notes:**

The SQL precompiler (or preprocessor) can be invoked via direct call (5250 command or CODE) or via option 14 in conjunction with a member type that indicates the source code includes embedded SQL (such as SQLRPGLE). The command to initiate the precompiler, Compile a member with embedded SQL (**CRTSQLRPGI**), automatically calls the RPG IV compiler, following successful preprocessing, unless the **\*NOGEN** option was selected. For ILE RPG source programs, if the **\*MODULE** option of **CRTSQLRPGI** is taken, the create an RPG ILE module (**CRTRPGMOD**) is automatically called instead of the create a bound RPG program (**CRTBNDRPG**) command.

The precompiler checks the syntax of the SQL statements, turns them into host language comments, and generates the host language statements to call the database manager.

Of note at this point is the fact that you must specify that whether you want a listing from the precompiler by entering:

Listing output . . . . . : OUTPUT \*PRINT

for the **OUTPUT** parameter.

The default is **\*NONE**.

The precompiler creates a source member (RPGLE) that is passed to the compiler. The compiler then processes this source member.

# Embedded SQL: Under the covers (1 of 2)

IBM i

```

000100  FApInv_PF  IF   E          K Disk
000200  FEmbedD1D  CF   E          Workstn InDDS(Indicators)
000300
000400  D Indicators    DS
000500  D Exit           1N  Overlay(Indicators:03)
000600  D Cancel         1N  Overlay(Indicators:12)
000700  D Error          1N  Overlay(Indicators:90)
000800
000900  D*      SQL COMMUNICATION AREA
001000 >>  D SQLCA       DS
001100  D SQLCAID        8A  INZ(X'0000000000000000')
001200  D SQLAID         8A  OVERLAY(SQLCAID)
001300  D SQLCABC        10I 0
001400  D SQLABC          9B 0 OVERLAY(SQLCABC)
001500 >>  D SQLCODE        10I 0
001600 >>  D SQLCOD         9B 0 OVERLAY(SQLCODE)
001700  D SQLERRML        5I 0
001800  D SQLERL          4B 0 OVERLAY(SQLERRML)
001900  D SQLERRMC        70A
002000  D SQLERM          70A OVERLAY(SQLERRMC)
002100  D SQLERRP          8A
002200  D SQLERP          8A OVERLAY(SQLERRP)
002300  D SQLERR          24A
002400  D SQLER1          9B 0 OVERLAY(SQLERR: *NEXT)
002500  D SQLER2          9B 0 OVERLAY(SQLERR: *NEXT)
002600  D SQLER3          9B 0 OVERLAY(SQLERR: *NEXT)
002700  D SQLER4          9B 0 OVERLAY(SQLERR: *NEXT)
002800  D SQLERS          9B 0 OVERLAY(SQLERR: *NEXT)
002900  D SQLER5          9B 0 OVERLAY(SQLERR: *NEXT)
003000  D SQLER6          9B 0 OVERLAY(SQLERR: *NEXT)
003100  D SQLERRD        10I 0 DIM(6) OVERLAY(SQLERR)
003200  D SQLWRN          1A
003300  D SQLWNN1        1A OVERLAY(SQLWRN: *NEXT)
003400  D SQLWNN2        1A OVERLAY(SQLWRN: *NEXT)
003500  D SQLWNN3        1A OVERLAY(SQLWRN: *NEXT)
003600  D SQLWNN4        1A OVERLAY(SQLWRN: *NEXT)
003700  D SQLWNN5        1A OVERLAY(SQLWRN: *NEXT)
003800  D SQLWNN6        1A OVERLAY(SQLWRN: *NEXT)
003900  D SQLWNN7        1A OVERLAY(SQLWRN: *NEXT)
004000  D SQLWNN8        1A OVERLAY(SQLWRN: *NEXT)
004100  D SQLWNN9        1A OVERLAY(SQLWRN: *NEXT)
004200  D SQLWNIA        1A OVERLAY(SQLWRN: *NEXT)
004300  D SQLWARN        1A DIM(11) OVERLAY(SQLWRN)
004400  D SQLSTATE        5A
004500  D SQLSTTT        5A OVERLAY(SQLSTATE)
004600  D* END OF SQLCA

```

© Copyright IBM Corporation 2011

Figure 5-57. Embedded SQL: Under the covers (1 of 2)

AS106.0

## Notes:

This example shows the RPG IV source member (RPGL E type) that is created by the precompiler. By default this source member is created in QTEMP unless you direct it to be stored elsewhere by changing the **TOSRCFILE** parameter from the default of **QTEMP**. The precompiler automatically includes the SQL Communications Area (SQLCA) data structure in the source. The **SQLCA** is used to obtain error information.

The notes are continued in the next visual.

## Embedded SQL: Under the covers (2 of 2)

IBM i

```

005500 >> D          DS
005600 >> D  SQL_00000      1    2B 0 INZ(128)
005700 >> D  SQL_00001      3    4B 0 INZ(1)
005800 >> D  SQL_00002      5    8B 0 INZ(0)
005900 >> D  SQL_00003      9    9A  INZ('0')
006000 >> D  SQL_00004     10   128A
006100 >> D  SQL_00005     129  132P 0 PACKEVEN
006200 >> D  SQL_00006     133  135P 0
006300 >> D  SQL_00007     136  143A
006400 >> D  SQL_00008     144  147P 0 PACKEVEN
006500
/FREE
006600   ExFmt Prompt;
006700   Dow Not Exit;
006800 // SQL I/O Statements
006900
007000 >> /*Exec SQL
007100 >> /* Select VndNbr, ApInvNbr, ApCheck#
007200 >> /* into :VndNbr, :ApInvNbr, :ApCheck#
007300 >> /* from ApInv_PF
007400 >> /* where PoNbr = :PoNbr;
007500 /END-FREE
007600 >> /*Exec SQL
007700 >> /* Select VndNbr, ApInvNbr, ApCheck#
007800 >> /* into :VndNbr, :ApInvNbr, :ApCheck#
007900 >> /* from ApInv_PF
008000 >> /* where PoNbr = :PoNbr;
008100 >> C          EVAL    SQL_00005  = PONBR
008200 >> C          Z-ADD   -4      SQLER6
008300 >> C          CALL    SQLROUTE
008400 >> C          PARM   SQLCA
008500 >> C          PARM   SQL_00000
008600 >> C          SQL_00003 IFEQ   '1'
008700 >> C          EVAL    VNDNBR = SQL_00006
008800 >> C          EVAL    APINVNBR = SQL_00007
008900 >> C          EVAL    APCHECK# = SQL_00008
00900 >> C          END
009100
/FREE
009200 // End SQL I/O Statements
009300 // Set Error indicator using SQLCOD

```

© Copyright IBM Corporation 2011

Figure 5-58. Embedded SQL: Under the covers (2 of 2)

AS106.0

### Notes:

The precompiler also:

- Builds a data structure that is designed to map any host variables used to parameters passed to the SQL DB2 processor.
- Comments out the SQL code, including the /EXEC SQL and /END-EXEC statements that frame the SQL statements.
- Inserts code that calls the SQL DB2 UDB (universal database) module, QSQROUTE.

The program variables, known as *SQL host variables*, are mapped to the data structure built by the precompiler. This data structure is passed as a parameter to QSQROUTE. In addition, the SQLCA is passed from QSQROUTE to your program.

# Taking advantage of embedded SQL

IBM i

- SQL has functions that are easily coded
  - Column functions: SUM, AVG
  - Scalar functions: ABS, SIN, COS
    - Rich date functions!
  - Conversion functions: CAST (anything to anything!)
- Can encapsulate SQL functions in RPG IV modules
- Packages encapsulated SQL functions in service programs
  - Write once, use everywhere
- SQL stored procedures enable client access to an i server
  - Stored procedure can encapsulate RPG IV program

© Copyright IBM Corporation 2011

Figure 5-59. Taking advantage of embedded SQL

AS106.0

## Notes:

We have seen that there are many functions available in APIs that cannot be performed as easily in RPG IV as those included in SQL. SQL also offers many functions that are coded very easily as embedded code within an RPG IV procedure or subprocedure. Some of the functions operate on columns (fields) and can produce a simple sum or an average. Others, known as *scalars*, operate on individual fields and can perform mathematical operations, such as finding a cosine, sine, or an absolute value of a numeric field.

One of the most flexible functions is CAST. With this function, SQL can convert from almost every data type to any other data type.

Many programmers would like even more date-related functions and BIFs in RPG IV. SQL provides these functions as part of the language!

Once the embedded SQL programs are written, you can make them available as encapsulated subprocedures that can be packaged in service programs.

SQL stored procedures can be compared to CL programs and can be called from an SQL interface. System i Navigator includes an interface called *SQL scripts*. Via this interface,

PC users can call SQL stored procedures on the i that can return data (known as *result sets*) to the PC. One form of SQL stored procedures is embedded SQL in an RPG IV program.

## Checkpoint (1 of 2)

IBM i

1. True or False: A notify object for restart information is required for implementing commitment control in your high-level language programs.
  
2. A trigger program:
  - a. Can be written in any IBM i high-level language
  - b. Is associated with a physical file
  - c. Is independent from applications
  - d. All of the above
  
3. The \_\_\_\_\_ command shows trigger information associated with a data file.
  - a. DSPFD
  - b. DSPFFD
  - c. DSPPGMREF
  - d. DSPPFM

© Copyright IBM Corporation 2011

Figure 5-60. Checkpoint (1 of 2)

AS106.0

### Notes:

## Checkpoint (2 of 2)

 IBM i

4. True or False: The application causing a trigger program to fire, waits for the trigger program to complete.
  
5. \_\_\_\_\_ embedded SQL statements are complete at compile time while \_\_\_\_\_ embedded SQL is completed, compiled, and executed during program execution.

© Copyright IBM Corporation 2011

---

Figure 5-61. Checkpoint (2 of 2)

AS106.0

### Notes:

## Unit summary

IBM i

Having completed this unit, you should be able to:

- Describe the elements of commitment control
- Implement basic commitment control in an RPG IV application
- Describe how trigger programs can be used to uniformly enforce business rules regarding a database.
- Activate a trigger program to a DB2 UDB file
- Code a trigger program, that is, a program to receive the trigger buffer from the system when a trigger program is fired.
- Describe the purpose of embedded SQL
- Recognize embedded SQL in an RPG IV program

© Copyright IBM Corporation 2011

Figure 5-62. Unit summary

AS106.0

### Notes:



# Unit 7. Advanced ILE topics

## What this unit is about

This unit describes the features of the Integrated Language Environment that we did not cover in AS07/AS070. These include multiple procedure modules and the NOMAIN H-spec keyword, using export for sharing data and modules. We also discuss more about binding and activation groups.

## What you should be able to do

After completing this unit, you should be able to:

- Create modules that contain more than one procedure or subprocedure
- Create NOMAIN modules to contain one or more subprocedures
- Explain the use of EXPORT for data and module sharing
- Create and use binding directories
- Use binder language for exporting data and modules
- Describe and use activation groups

## How you will check your progress

Accountability:

- Machine exercises
- Checkpoint questions

## Unit objectives

IBM i

After completing this unit, you should be able to:

- Create modules that contain more than one procedure or subprocedure
- Create NOMAIN modules to contain one or more subprocedures
- Explain the use of EXPORT for data and module sharing
- Create and use binding directories
- Use binder language for exporting data and modules
- Describe and use activation groups

© Copyright IBM Corporation 2011

Figure 6-1. Unit objectives

AS106.0

### Notes:

## 7.1. Packaging modules

# Review

IBM i

- What you know:
  - Prototyped CallP
  - Bind by copy
  - Bind by reference and service programs
  - Basic export of data from subprocedures
  - Procedures, subprocedures, modules

© Copyright IBM Corporation 2011

Figure 6-2. Review

AS106.0

## Notes:

By this time, you should have a good understanding of binding in the ILE world. You should be able to bind several modules together to create an executable program using bind by copy or bind by reference to a service program, or using a combination of both types of binding.

Now we look at a more complex environment and discuss binding many modules, using the NOMAIN H-spec keyword to package many procedures in the same module.

You have learned that the EXPORT keyword can be used to make data items available outside an individual module. The EXPORT keyword also applies to procedures. You can use it to make a procedure or subprocedure callable outside its module container.

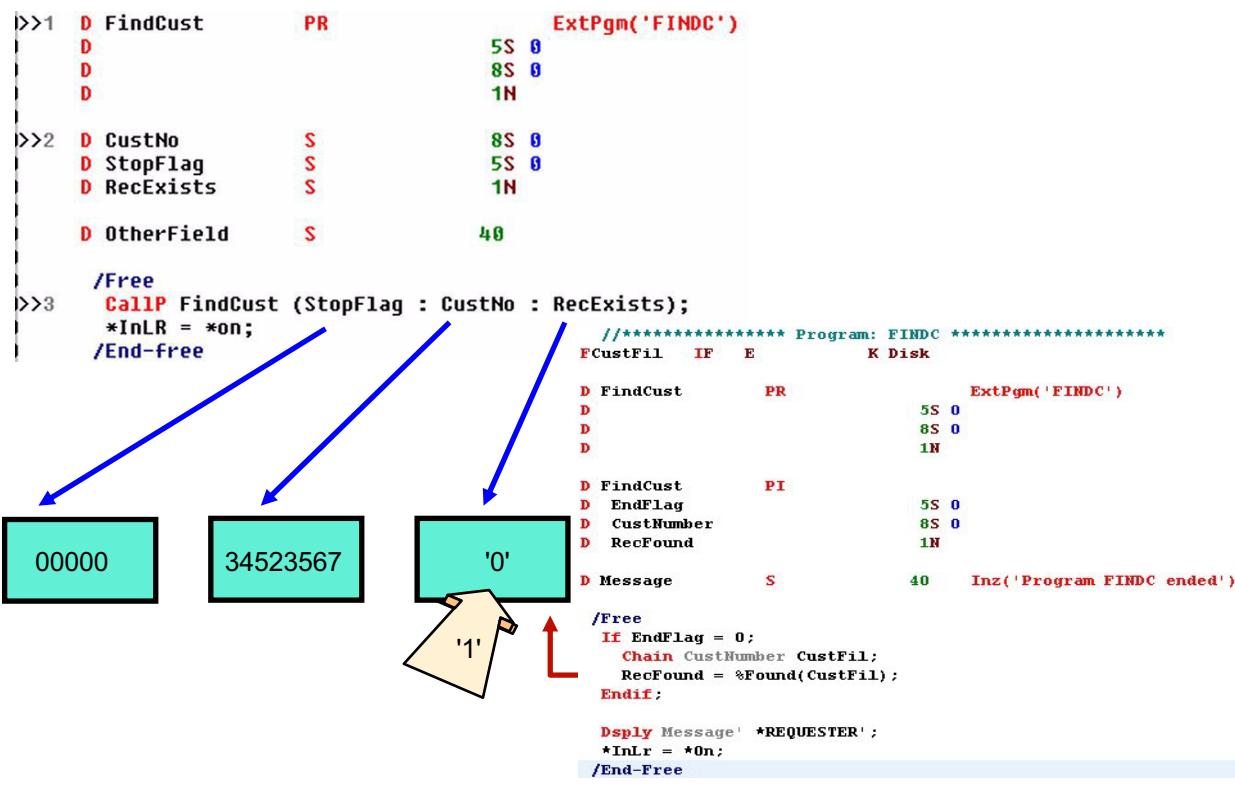
Binding directories and the binder language are particularly useful when creating applications. They can be especially useful in the maintenance phase of application development.

ILE activation groups are used to isolate resources. They are often misunderstood and can cause problems when you create ILE objects and neglect to specify an activation group.

In the previous unit, you saw an introduction to embedded SQL. In this unit, we also show you how to encapsulate embedded SQL PRG IV programs as functions that can be called by other modules.

# Calling program calls called program

IBM i



© Copyright IBM Corporation 2011

Figure 6-3. Calling program calls called program

AS106.0

## 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.

# CONST and OPTIONS keywords 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)
- CONST
  - Parameter treated as read-only
  - Can pass the result of expression as a parameter
  - Data type and size may not exactly match

© Copyright IBM Corporation 2011

Figure 6-4. CONST and OPTIONS keywords for parameter definition

AS106.0

## 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*.

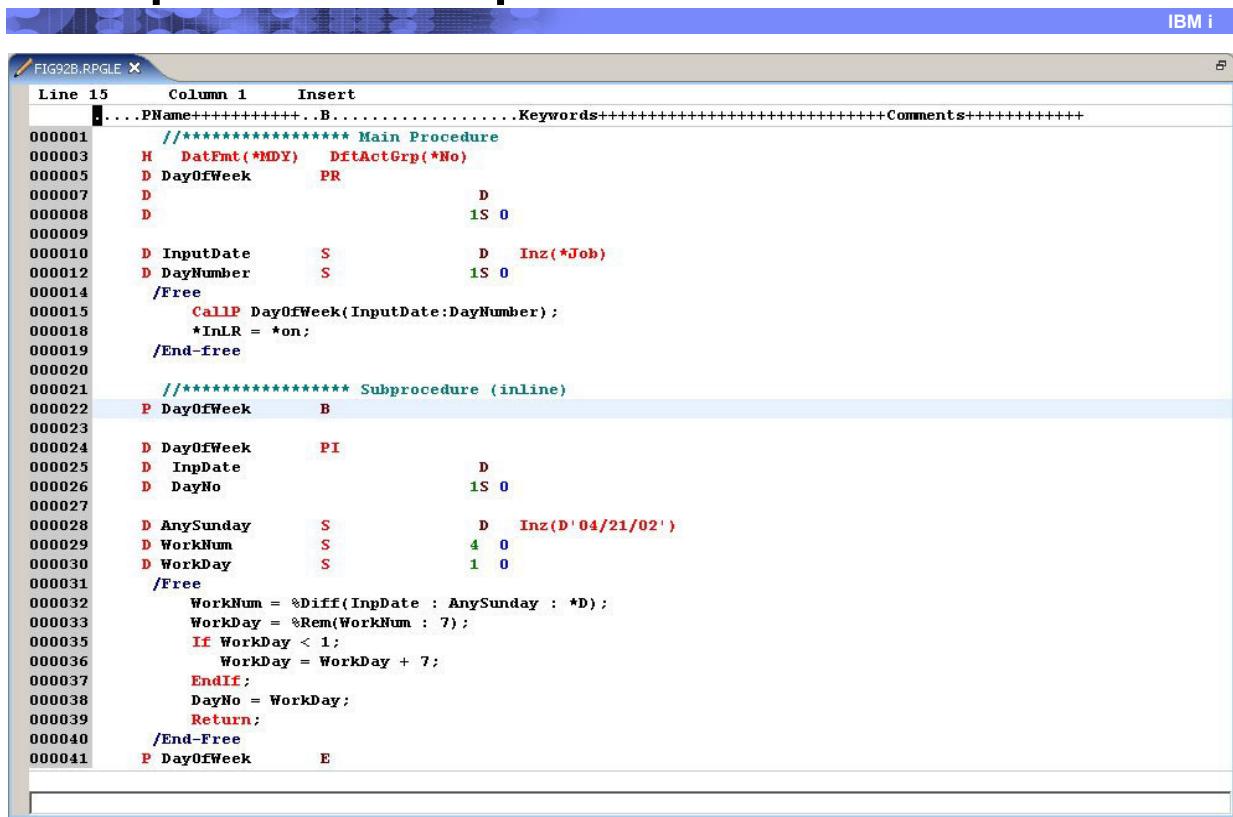
\* OMIT is only allowed for parameters that are passed by reference, including those that specify the CONST keyword.

- \* 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).

- CONST tells the compiler to accommodate possible mismatches in the definition of the parameters in the called program versus the calling program. This keyword also allows the result of an expression to be passed as a parameter.

# 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
000010 D InputDate     S      D  Inz(*Job)
000012 D DayNumber     S      1S 0
000014 /Free
000015     CallP DayOfWeek(InputDate:DayNumber);
000018     *InLR = *on;
000019 /End-free
000020
000021 //***** Subprocedure (inline)
000022 P DayOfWeek     B
000023
000024 D DayOfWeek     PI
000025 D InpDate       D
000026 D DayNo         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(InpDate : AnySunday : *D);
000033     WorkDay = %Rem(WorkNum : 7);
000035     If WorkDay < 1;
000036         WorkDay = WorkDay + 7;
000037     EndIf;
000038     DayNo = WorkDay;
000039     Return;
000040 /End-Free
000041 P DayOfWeek     E

```

© Copyright IBM Corporation 2011

Figure 6-5. Example: Basic subprocedure

AS106.0

## Notes:

Notice the sequence of the specifications in the visual. In this example, the subprocedure is coded in line following the main procedure. Let's review the main components of subprocedure coding.

In summary:

- The prototypes for the subprocedure are coded first. The main procedure calls the subprocedure. Therefore, a prototype (PR) that matches the procedure interface (PI) for the subprocedure is required.
- A call to the subprocedure is made using the CALLP operation.
- A P-specification that marks the beginning of a subprocedure with the character 'B' in the appropriate column is required. Another P-specification that marks the end of a subprocedure with the character 'E' in the appropriate column is also required. P-specs (in pairs) appear after the regular C-specs.
- 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.
- Logic for the subprocedure.
- A return to the caller.
- A P-spec to end the subprocedure.

# Modular programming

IBM i

- Writing code once, using in many applications
- Writing smaller procedures and subprocedures
  - Can be easier to maintain than large programs
  - Can purchase / use commercially available routines
  - Can put working modules into production sooner
- Using flexible production environment
- Static binding
  - Best CALL performance
  - Prototyped CALLs
  - Encourages and enables modular application design

© Copyright IBM Corporation 2011

Figure 6-6. Modular programming

AS106.0

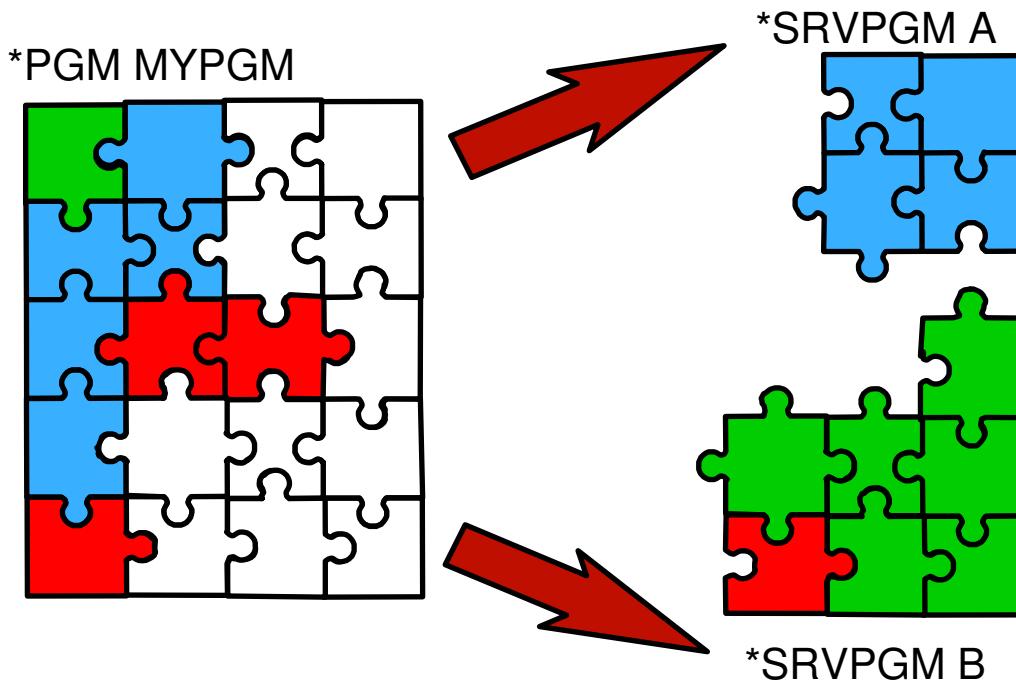
## Notes:

Leveraging the features of ILE is more easily done if your code is modular. Procedures and subprocedures tend to focus on a single function or task. Because of this focus, code tends to become simpler to write, and, as a result, can be put into production more quickly. Modular code can also be more easily maintained than traditional large programs.

In order to support this environment, ILE provides tools to enable access to code and data from many applications.

# Program with many modules

IBM i



© Copyright IBM Corporation 2011

Figure 6-7. Program with many modules

AS106.0

## Notes:

You recall that a single ILE program can be composed of one or many modules.

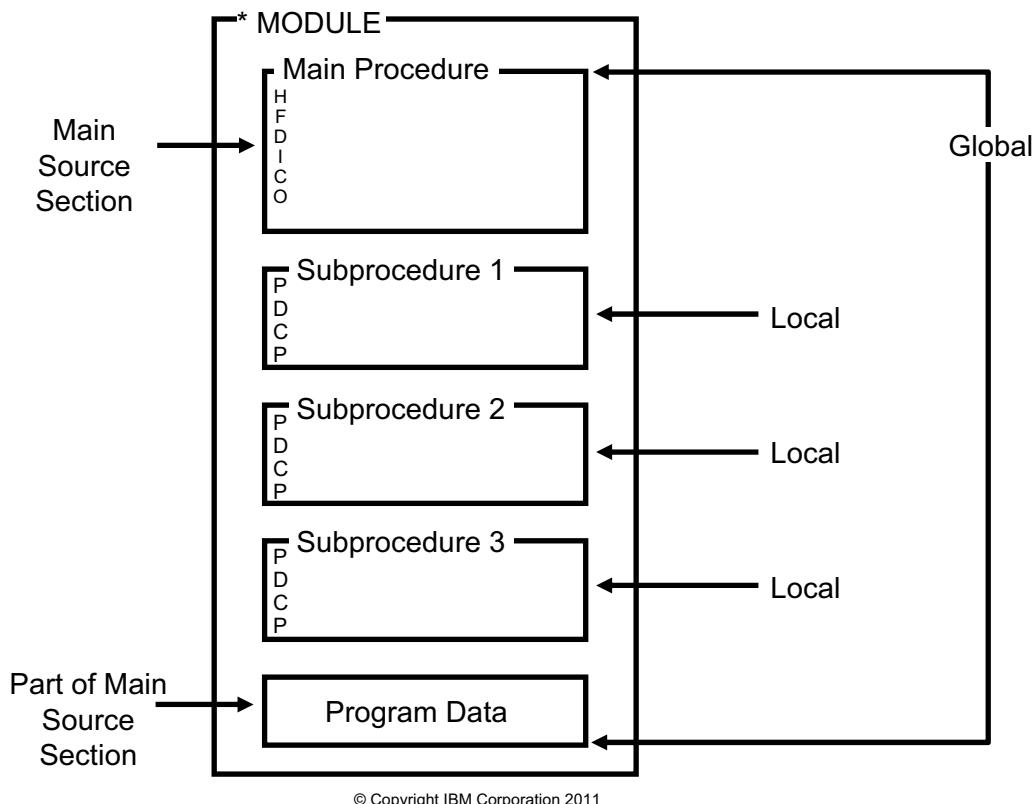
In this visual, the ILE program has many modules bound by copy and many other modules bound by reference (to two different service programs).

When MYPGM is called, one of the steps in activating the program is to connect to the two service programs, A and B. This connection is very similar to, and takes about the same time as, a dynamic program call. The connection between \*PGM MYPGM and \*SRVPGM A and \*SRVPGM B happens as part of the activation of the \*PGM MYPGM.

We have always shown you examples where a single module contains only one procedure. However, it is also possible to code more than one procedure, usually subprocedures in a single source member, and then package them in a single module.

# Module can contain more than one procedure

IBM i



© Copyright IBM Corporation 2011

Figure 6-8. Module can contain more than one procedure

AS106.0

## Notes:

An ILE RPG module may contain one or more RPG IV source members. The procedures that can make up an ILE RPG module are:

- An *optional main procedure* which consists of the set of H, F, D, I, C, and O-specifications that begin the source. This is the like the *normal (or traditional)* procedure that we have been writing in our classes so far.
- *Zero or more subprocedures*, which are coded on P, D, and C-specifications.

Subprocedures do not use the RPG cycle. A subprocedure may have local variables that are available for use only by the subprocedure itself.

The main procedure (if coded) can always be called by other modules packaged in the ILE program object. Subprocedures are local to the module in which they are contained and may be called from that module. The subprocedures contained in a module with a main procedure can be made available to other modules using *export* (we will discuss export in more detail later).

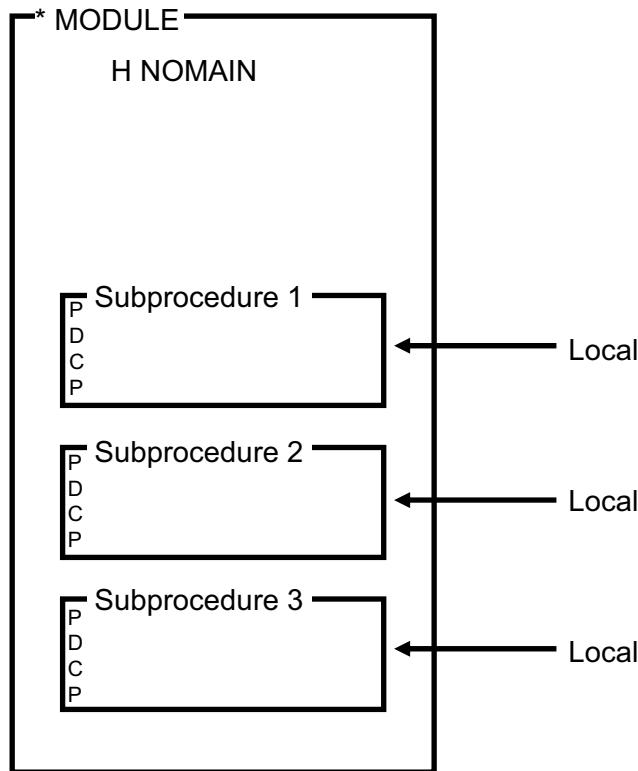
If the subprocedures are local, they can only be called by the main procedure or the other subprocedures in the module; if they are exported from the module, they can be called by any procedure in the program.

Subprocedures may also be packaged as a group in a very convenient module package using the NOMAIN keyword on the H-spec.

Module creation consists of compiling a source member, and, if that is successful, creating a \*MODULE object. The \*MODULE object includes a list of imports and exports referenced within the module. It also includes debug data if you request this at compile time.

# Subprocedures with NOMAIN procedure

IBM i



© Copyright IBM Corporation 2011

Figure 6-9. Subprocedures with NOMAIN procedure

AS106.0

## Notes:

A main procedure is not mandatory. If all you want to do is create a module containing a group of related subprocedures, you may do so by creating a module specifying NOMAIN on the H-spec. This module cannot be used to create a \*PGM object. It can be bound by copy or reference to other modules that contain a main procedure. The best place to package NOMAIN modules is in a service program.

This method is useful when you want to group either functionally related or application-related subprocedures together.

Because there is no main procedure in a NOMAIN module, the ILE module that you create cannot be called. It has no RPG cycle logic in it, and therefore you have less overhead in the module.

# Today's RPG IV program

IBM i

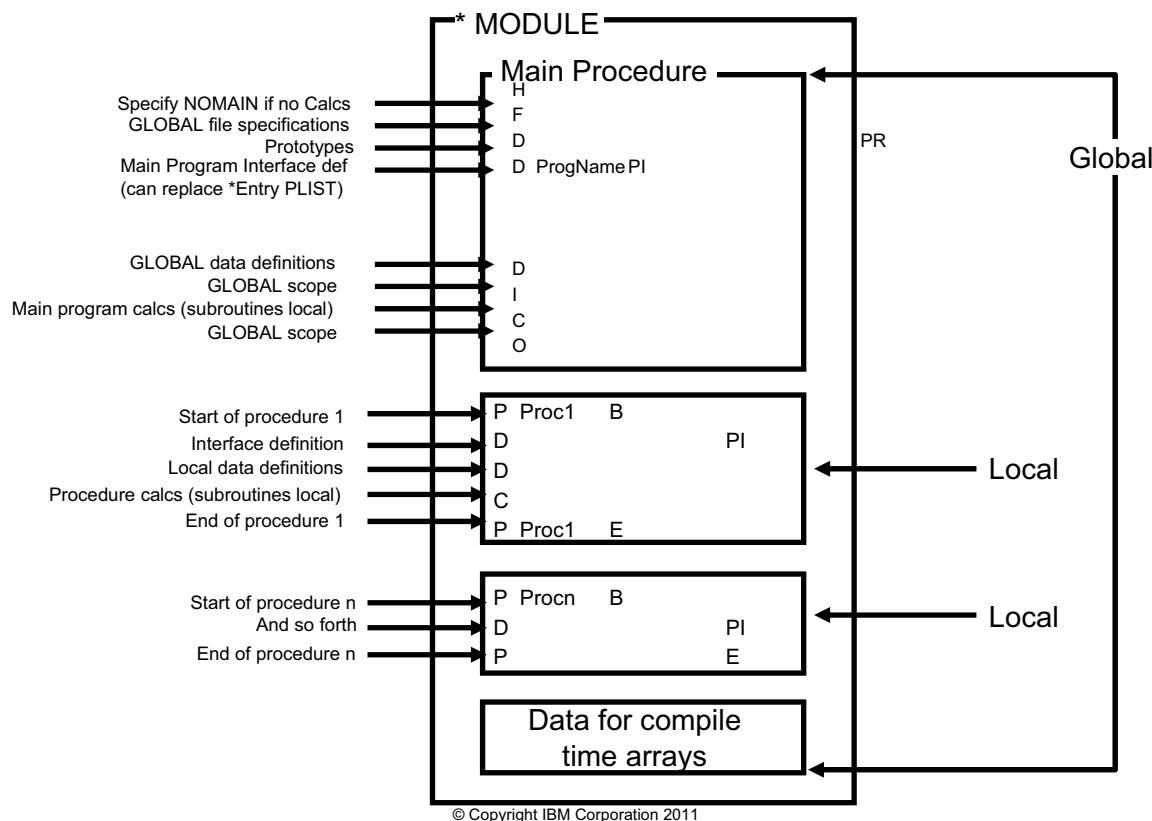


Figure 6-10. Today's RPG IV program

AS106.0

## Notes:

This visual illustrates all the parts that can be included in an RPG module to allow you to call a subprocedure within the module from another program. You can use this visual to remind you of the structure of a module that includes one (or many) subprocedures.

Because this *program* uses the ILE facility of subprocedures, it cannot be bound to run in the default activation group. You cannot specify **\*YES** for the **DFTACTGRP** parameter of **CRTBNDRPG**. If you try to bind to run in the **DFTACTGRP**, the compile will fail.

# EXPORT keyword on P-specification

IBM i

- Makes procedure or subprocedure available to be called from outside the \*MODULE which contains it
- Coded on the P-specification
- Makes procedures available for bind by reference and bind by copy

© Copyright IBM Corporation 2011

Figure 6-11. EXPORT keyword on P-specification

AS106.0

## Notes:

When you *export* a procedure in ILE, you make it available to be called by procedures outside this module object. An export can also be called a *definition*. A procedure is made available for export when you specifically use the export keyword in the prototype.

If the EXPORT keyword is not specified, the procedure or subprocedure can only be called from within the module in which it is contained. Remember that a module can contain many subprocedures.

*Procedures and subprocedure names are not imported using the IMPORT keyword.* The IMPORT keyword is used for data items only. *Procedures and subprocedures are imported implicitly* by any module in the program that makes a bound call to the procedure or that uses the procedure name to initialize a procedure pointer.

# Subprocedure packaged in same module as main procedure: Example

IBM i

```

FAgeInq   CF   E           Workstn IndDS(WKIND)
D WkInd    DS
D Exit     3      3N
D BadDate  40     40N
D BornMonth S      2  0
D CurrMonth S      2  0

>>2 DNbrDays   PR      5  0
/FREE
  Write Header;
  Write Footer;
  ExFmt I Prompt;

  Dow NOT Exit;
  // Test for valid date value
  Test(DE) Born;
  If %error;
    BadDate = *On;
  Else;
    // Display details
    Age = %Diff(%date(*date):%date(Born):*y);
    Months = %Diff(%date(*date):%date(Born):*m) - Age*12;
    // Call NbrDays to determine # days since Jan. 1, 2001
  >>3   NoDays = NbrDays;
         Write Detail;
         EndIf;
  // Display prompt
  ExFmt Prompt;
Enddo;

*InLR = *On;
Return;
/End-free
//*****End of Main Procedure*****
>>1 PNbrDays   B
>>2 DNbrDays   PI      5  0
/FREE
  Return %DIFF(%DATE(*DATE):D'2001-01-01':*D);
  *InLR = *On;
/END-FREE
P                   E

```

© Copyright IBM Corporation 2011

Figure 6-12. Subprocedure packaged in same module as main procedure: Example

AS106.0

## Notes:

This code is made up of two components: a main procedure, (AgeDemo), and a subprocedure (NbrDays).

Notice the following aspects:

1. The subprocedure is delimited with P-specifications.
2. The subprocedure passes one return parameter. Notice the simplicity of the PI and the PR. And, even though the subprocedure is part of the module that contains Agedemo, you must code the PR in the main procedure since it will call the subprocedure.
3. Even though this looks like a simple assignment statement, it is really an expression call to the NbrDays subprocedure.
4. The subprocedure calculates the number of days since January 1, 2001, and returns it to the caller.

# Call NbrDays from another procedure

IBM i

```

000001 // Declare Files
000002 F Item PF IF E K Disk
000003 F ItemInq2 CF E Workstn IndDS(WkstnInd)
000004
000005 // Map indicators in DSPF to named indicators
000006 D WkstnInd DS
000007 D NotFound 40 4ON
000008 D LowQty 30 3ON
000009 D Exit 03 03N
000010
000011
000012 >>1 DNbrDays PR 5 0
000013
000014 /FREE
000015 PgmName = 'ITEMINQOV';
000016 // Call NbrDays to determine # days since Jan. 1, 2001
000017 >>2 NcDays = NbrDays;
000018
000019 Write Header; // Write to buffer
000020 Write Footer; // Write to buffer
000021 Exfmt Prompt; // Write to buffer; write buffer to display; enable read
000022
000023 Dow NOT Exit;
000024 Chain ItmNbr Item_PF;
000025 NotFound = Not %found(Item_PF); // Set indicator for record not found
000026
000027 If %found(Item_PF); // Item Number valid?
000028 LowQty = (ItmQtyOH + ItmQtyOO) < 20; // Set Indicator for Qty < Minimum
000029 Write Detail; // Write to buffer
000030 Endif;
000031
000032 // Display prompt with error or not
000033 Exfmt Prompt; // Write to buffer; write buffer to display; enable read
000034 Enddo;
000035 // F3 pressed; user wants to exit program
000036 *InLR = *on;
000037 /END-FREE

```

Error Message on compile listing:

Errors were found during the binding step. See the job log for more information.

Job Log:

Message . . . : Definition not found for symbol 'NBRDAYS'.

© Copyright IBM Corporation 2011

Figure 6-13. Call NbrDays from another procedure

AS106.0

## Notes:

We added code to call the NbrDays subprocessure from the ITEMINQ2 procedure. The \*PGM could not be created.

Why?

The steps we followed:

1. **CRTRPGMOD ItemInq2.**
2. **CRTRPGMOD AgeDemo.** Because we want to bind a copy of the NbrDays subprocessure, we need a module.
3. **CRTPGM ItemInq** binding by copy modules ItemInq2 and AgeDemo.

## Add EXPORT to NbrDays

```

FAgeInq  CF   E          Workstn  IndDS(WKIND)
D WkInd
D Exit      DS           3        3N
D BadDate    40           40N
D BornMonth  S            2  0
D CurrMonth  S            2  0

DNbrDays     PR           5  0
/Free
  Write   Header;
  Write   Footer;
  ExFmt   Prompt;

  Dow NOT Exit;
  // Test for valid date value
  Test(DE) Born;
  If %error;
    BadDate = *On;
  Else;
    // Display details
    Age = %Diff(%date(*date):%date(Born):*y);
    Months = %Diff(%date(*date):%date(Born):*m) - Age*12;
    // Call NbrDays to determine # days since Jan. 1, 2001
    NoDays = NbrDays;
    Write Detail;
  EndIf;
  // Display prompt
  ExFmt Prompt;
Enddo;

*InLR = *On;
Return;
/End-Free
//*****End of Main Procedure*****
>>1 PNbrDays     B           Export <<1
DNbrDays     PI           5  0
/FREE
  Return %Diff(%date(*date):D'2001-01-01':*D);
  *InLR = *on;
/END-FREE
P             E

```

© Copyright IBM Corporation 2011

CRTRPGMOD AgeDemo  
No changes to ItemInq2  
CRTPGM  
PGM(AS10V6LIB/ITEMINQ)

MODULE(AS10V6LIB/ItemInq2  
AS10V6LIB/AgeDemo)

Figure 6-14. Add EXPORT to NbrDays

AS106.0

### Notes:

Once we make the change to AgeDemo by specifying (through EXPORT) that we want to make the NbrDays subprocedure available to other modules, we:

1. Recompile the AgeDemo module.
2. Run the **CRTPGM** command again to re-create a program named **ItemInq**.

Because the NbrDays symbol can now be resolved at bind time, during the **CRTPGM**, the program is created.

# Packaging subprocedures in a NOMAIN procedure

IBM i

- No RPG logic cycle overhead
- Faster execution
- Efficient packaging in service program
- Use /COPY and conditional compiler directives for prototypes
- DFTACTGRP = \*NO
- Remember to EXPORT subprocedures to enable call from NOMAIN module
- Cannot call NOMAIN procedure (no entry point)

© Copyright IBM Corporation 2011

Figure 6-15. Packaging subprocedures in a NOMAIN procedure

AS106.0

## Notes:

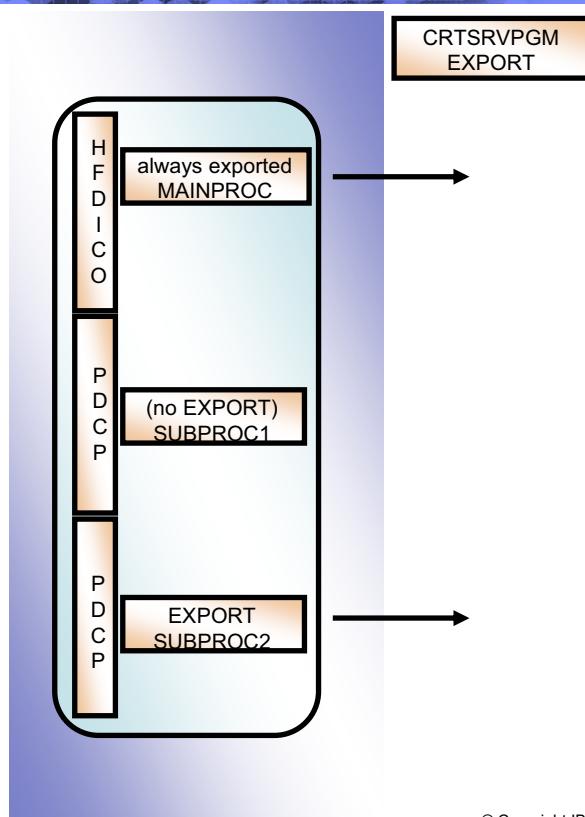
A NOMAIN module consists only of subprocedures; there is *no main procedure*. A NOMAIN module executes faster and requires less storage because there is no cycle code that is created for the module. You specify a NOMAIN module by coding the NOMAIN keyword on the control specification.

A NOMAIN module is a perfect means of packaging one or many subprocedures. These modules may be bound by copy or, better still, contained in a service program and bound by reference to the calling procedures.

When you create a NOMAIN procedure, you must consider issues such as EXPORT and how to group subprocedures in NOMAIN modules. Also, if you package any number of subprocedures in a NOMAIN procedure, you will have to code their prototypes in the calling procedures. The best way to accomplish this task is to use /COPY and conditional compiler directives.

# Export considerations

IBM i



- **EXPORT keyword in CRTSRVPGM:**
- Determines which \*MODULE exports outside \*SRVPGM
  - \*ALL = MAINPROC + SUBPROC2
  - \*SRCFILE is selective: Can export only SUBPROC2
- \* SUBPROC1 cannot be made available (no EXPORT on P-spec)

© Copyright IBM Corporation 2011

Figure 6-16. Export considerations

AS106.0

## Notes:

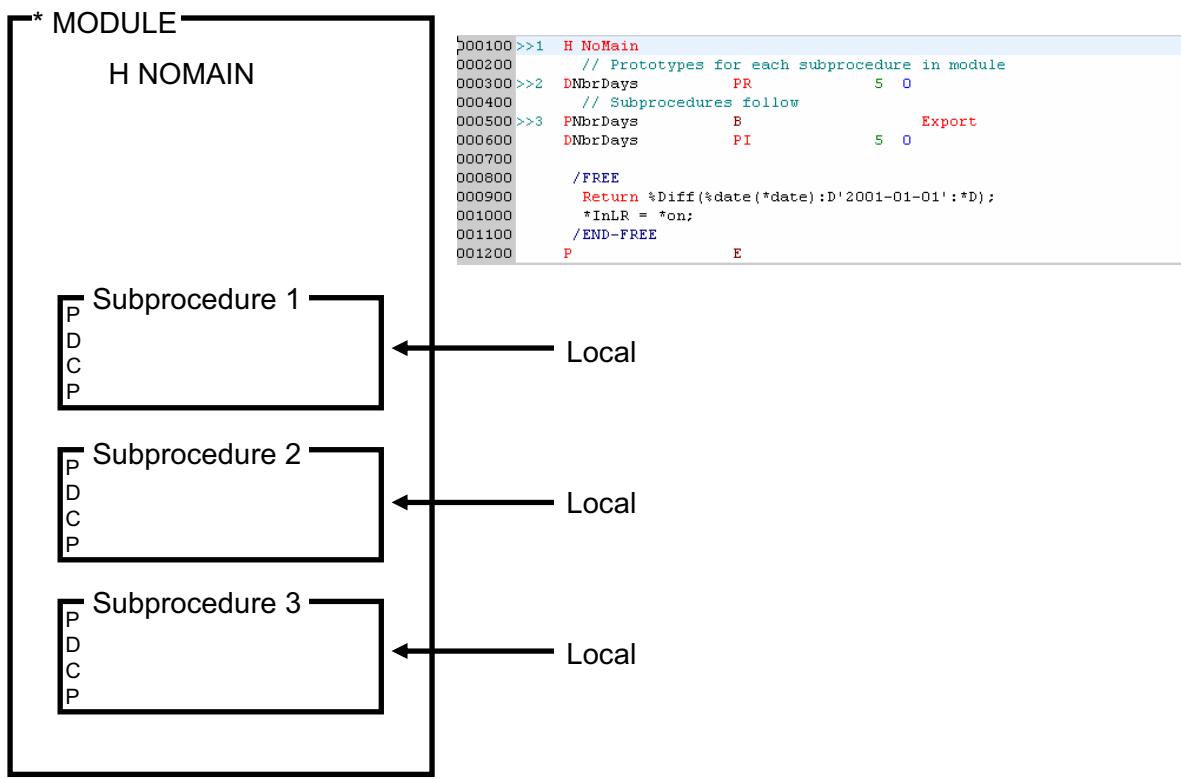
At first glance, the whole process of exporting procedures might seem confusing.

Only those procedures that specify the EXPORT keyword on P-spec are exported beyond the \*MODULE. Binder language and **CRTSRVPGM** have nothing to do with this.

The **CRTSRVPGM EXPORT** parameter coupled with binder language control exports (data and procedure) beyond the \*SRVPGM. These two functions can only export items which are available when the EXPORT keyword is specified on the D-spec for data and the P-spec for procedures.

# Subprocedures in a NOMAIN procedure

IBM i



© Copyright IBM Corporation 2011

Figure 6-17. Subprocedures in a NOMAIN procedure

AS106.0

## Notes:

We took the NbrDays subprocedure and copied it from the AgeDemo procedure. We then deleted the subprocedure from the AgeDemo procedure.

In order to package the subprocedure in the NOMAIN procedure, we:

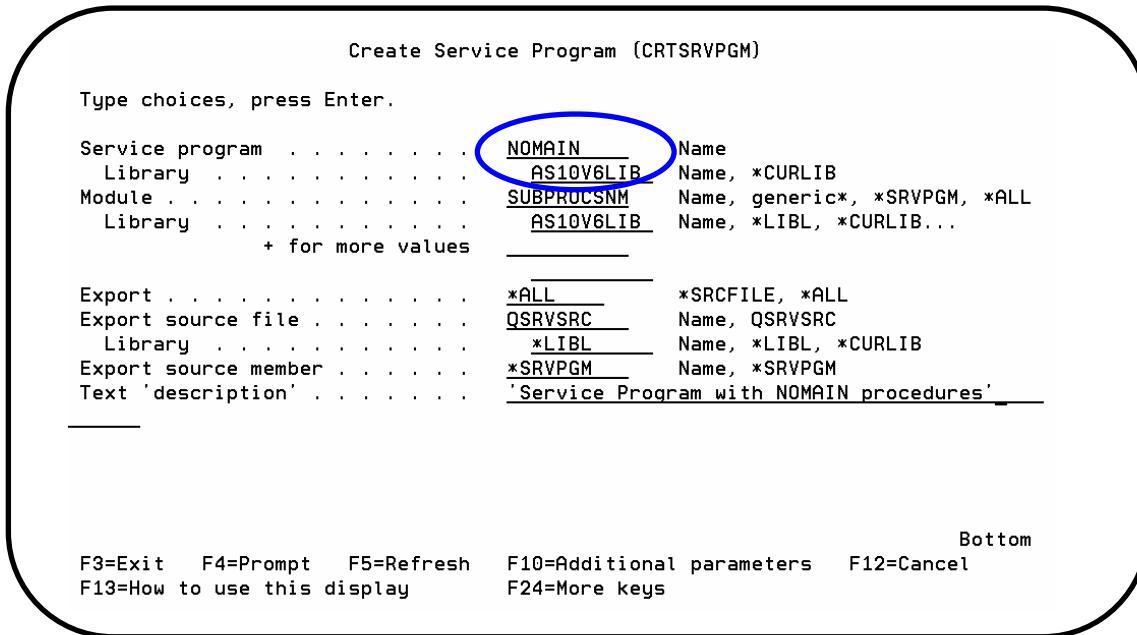
1. Added an H-spec with the NOMAIN keyword. No RPG logic cycle overhead is included in this procedure.
2. Copied the prototype (PR) for the subprocedure to the NOMAIN procedure. We would do this for each and every subprocedure packaged in the module, just as we would do when packaging subprocedures in a main procedure. The rules for prototypes are consistent. If you like, think of the prototypes in the NOMAIN procedure as the 'keys' to the subprocedures contained in the module.
3. Notice that you must still specify EXPORT for each subprocedure that you want known outside the NOMAIN module. Of course, you probably want to export all subprocedures. You can specify more information about specific exports when you create the service program.

The process shown in this visual would be repeated for each subprocedure to be included in the service program.

# Steps required to make subprocedures available

IBM i

- **CTRPGMOD SUBPROCSNM** (holds NbrDays)
- **CRTSRVPGM NOMAIN**



© Copyright IBM Corporation 2011

Figure 6-18. Steps required to make subprocedures available

AS106.0

## Notes:

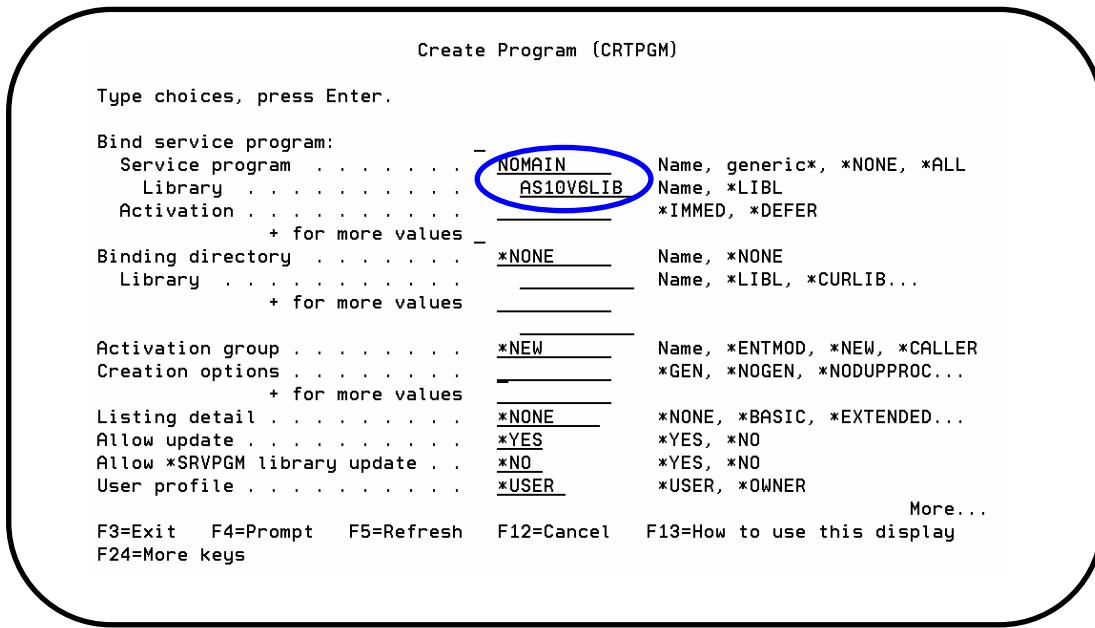
Working with our AgeDemo program, we copy the subprocedure NbrDays to a new member that we name SUBPROCSNM. We then create a module and then create a service program named NOMAIN that (for now) holds only this one module.

Notice that we have some choices regarding exporting of both data items and subprocedures. For now, we simply specify **\*ALL**, but we discuss binder language later in this unit.

# Bind subprocedures in service program

IBM i

1. CRTRPGMOD AGEDEMO (with NbrDays removed)
2. CRTPGM AgeDemoMN



© Copyright IBM Corporation 2011

Figure 6-19. Bind subprocedures in service program

AS106.0

## Notes:

This visual shows the steps to bind by reference the NbrDays module in the NOMAIN service program:

1. The NbrDays code is deleted from the AgeDemo procedure, and we re-create the AgeDemo module.
2. We create a new \*PGM named AGEDEMOMN that binds by reference the NbrDays object in the NOMAIN service program.

# Summary of application

IBM i

- Module AgeDemoMN:
  - Contains main procedure AgeDemoMN
  - Calls NbrDays subprocedure
- Module SubProcsNM
  - Contains subprocedure NbrDays
  - Cannot be called
- Service program NOMAIN
  - Contains module SubProcsNM
- Program AgeDemoNM
  - Contains module AgeDemoNM (entry procedure)
  - Binds by reference to NOMAIN \*SRVPGM

© Copyright IBM Corporation 2011

Figure 6-20. Summary of application

AS106.0

## Notes:

At this point, we should summarize what we have so far. We have used the features of ILE to package our code in a much more modular fashion. Even though all we have done so far is to package one subprocedure in a service program, we are ready to add more code to the service program.

# Machine exercise: Enhancing Nomain service program

IBM i



© Copyright IBM Corporation 2011

Figure 6-21. Machine exercise: Enhancing Nomain service program

AS106.0

## Notes:

Perform the machine exercise *enhancing Nomain service program*.

## 7.2. Binder language

## What is binder language?

IBM i

- Used to list exports of data and procedures for a service program
- Item listed for export *must* be specifically exported using the EXPORT keyword or bind will fail
- Create/edit source member in QSRVSR (default) or file name you choose
  - Supported by LPEX and SEU editors

© Copyright IBM Corporation 2011

Figure 6-22. What is binder language?

AS106.0

### Notes:

IBM i offers you several ways to make the process of specifying which modules to bind and which procedures and data items are to be exported and made available to other procedures not contained in this specific service program.

Binder language:

- Is a small set of nonrunnable commands that defines the exports for a service program.
- Enables you to itemize those subprocedures and data items that you want to export from the service program that contains the subprocedures. You would use the binder language rather than specifying \*ALL for the **EXPORT** parameter or **CRTSRVPGM** or **UPDSRVPGM**. Note that any items that you do not export are available via call from one subprocedure to another within the service program only.
- Enables the Source Entry Utility (SEU) syntax checker to prompt and validate the input when a BND source type is specified.

**Note:** You cannot use the SEU syntax checking type BND for a binder source file that contains wildcarding. You also cannot use it for a binder source file that contains names longer than 254 characters.

There is no format for SEU type BND.

If either of the following conditions exists, you do not need to use the binder language:

- A service program that never changes
- Users of the service program who do not mind changing their programs when a signature changes

However, because this situation is not likely for most applications, you should always use binder language for all service programs.

# Using binder language

IBM i

```

FINRTN.BND X
Line 1      Column 1      Replace
000100      StrPgmExp PgmLvl(*current) lvlchk(*yes)
000101      Export symbol(term)
000102      Export symbol(rate)
000103      Export symbol(amount)
000104      Export Symbol(openAcct)
000105      Export Symbol(closAcct)
000106      EndPgmExp

```

Change the **CRTSRVPGM** command to reference the binder source member:

```

CRTSVRPGM SRVPGM(FINRTN) . . . . EXPORT(*SRCFILE)
SRCFILE(QSRVSRC) SRCMBR(*SRVPGM)

```

© Copyright IBM Corporation 2011

Figure 6-23. Using binder language

AS106.0

## Notes:

Binder language makes subprocedures, procedures, and data names available for use by other \*PGM and \*SRVPGM objects by making them available for export.

Binder language consists of these following commands:

1. A Start Program Export (**STRPGMEXP**) command, which identifies the beginning of a list of exports from a service program.
2. One or more Export Symbol (**EXPORT**) commands, each of which identifies a data or procedure (module) symbol name available to be exported from a service program.
3. An End Program Export (**ENDPGMEXP**) command, which identifies the end of a list of exports from a service program.

*Only those symbols that are specifically exported using the EXPORT keyword on D-specs or P-specs are exported. Simply specifying the symbol name on an EXPORT statement is not enough.*

Notice the **CRTSRVPGM** command references the binder source member.

In this visual, we have created the \*SRVPGM and specified that we use the binder language to specifically list those procedures and data items that we want to make available outside the \*SRVPGM by specifying export.

Binder language is very simple but is created and saved as source only. It is used when creating a service program to determine what is to be available for export from a service program.

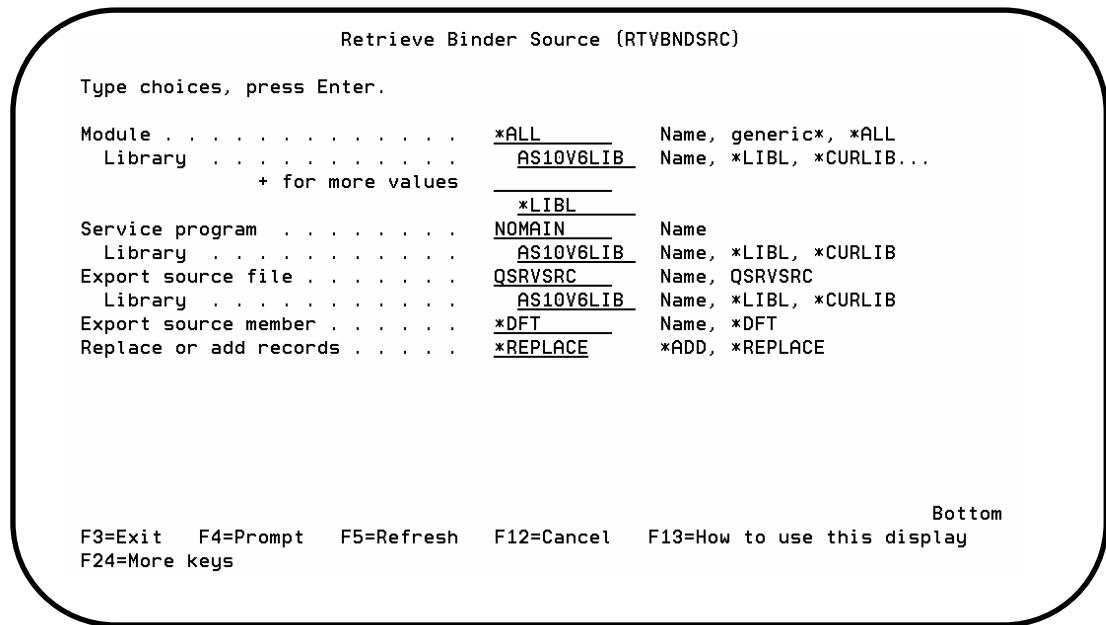
The code on the visual is stored in a source member type BND in source file (QSRVSRC). The binder source file is a parameter of the **CRTSRVPGM** or **UPDSRVPGM** commands. In both the **CRTSRVPGM** and the **UPDSRVPGM** commands, you may specify that binder language source is to be used to define procedures that are to be exported.

The export source file parameter should contain the name of the source member that contains the binder language listing those procedures and data elements that you want to make available for binding to other \*PGM or \*MODULE objects.

We cover the **LVLCHK** parameter (which refers to the signature of the service program) of **STRPGMEXP** in a few visuals from now.

# RTVBNDSRC

IBM i



© Copyright IBM Corporation 2011

Figure 6-24. RTVBNDSRC

AS106.0

## Notes:

Another useful command is Retrieve Binder Source, (**RTVBNDSRC**). This command can be used to help you to generate the binder language source based upon the exports from one or more modules.

# Output of RTVBNDSRC

IBM i

```

0000.01 STRPGMEXP PGMLVL(*CURRENT) SIGNATURE(X'0000000000000000E2E8C1C4D9C2D5
0000.02 /* **** */
0000.03 /* *SRVPGM NOMAIN AS10V6LIB 07/29/11 03:00:36 */
0000.04 /* **** */
0000.05 EXPORT SYMBOL("NBRDAYS")
0000.06 /* **** */
0000.07 /* *MODULE AGEDEMO AS10V6LIB 07/29/11 03:00:36 */
0000.08 /* **** */
0000.09 EXPORT SYMBOL("NBRDAYS")
0000.10 EXPORT SYMBOL("AGEDEMO")
0000.11 /* **** */
0000.12 /* *MODULE AGEDEMOMN AS10V6LIB 07/29/11 03:00:36 */
0000.13 /* **** */
0000.14 EXPORT SYMBOL("AGEDEMOMN")
0000.15 /* **** */
0000.16 /* *MODULE APINQSUB AS10V6LIB 07/29/11 03:00:36 */

```

© Copyright IBM Corporation 2011

Figure 6-25. Output of RTVBNDSRC

AS106.0

## Notes:

This is the member that is created when the command on the previous visual is executed. The **RTVBNDSRC** command examines the exported symbols of specifically selected or all modules in the library. The main things you need to be concerned about are the modules to be selected and the name of the source member. This name can be based on the name of the service program to be created (if you specify a name) or a specific name that you decide. Note that this process simply generates source code that you must then edit. No service program is generated. You would run **CRTSRVPGM** once you had edited the binder source.

Notice the generated quotations around the symbols. If the exported symbols contain lowercase letters, the symbol name should be enclosed within apostrophes. If apostrophes are not used, the symbol name is converted to all uppercase letters.

In the example below, the binder searches for an export named **P1**, not **p1**. It also searches for symbols **p2** and **p3**:

```
EXPORT SYMBOL(p1)
EXPORT SYMBOL('p2')
EXPORT SYMBOL('p3')
```

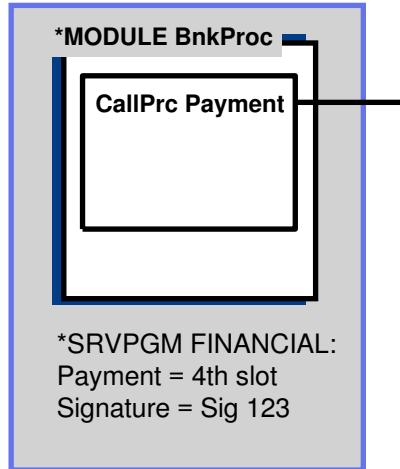
During the edit process, you would change the quotes as necessary.

The file created by the **RTVBNDSRC** command contains all symbols eligible to be exported from the modules, specified in the binder language syntax. *You should edit this file to include only the symbols you want to export*, then specify this file on the **SRCFILE** parameter of the **CRTSRVPGM** command.

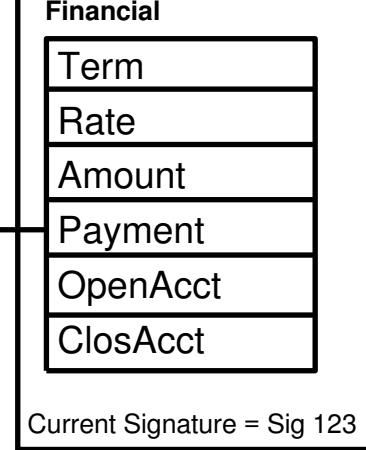
# Signature of service program

IBM i

## \*PGM Banker



## \*SRVPGM Financial



© Copyright IBM Corporation 2011

Figure 6-26. Signature of service program

AS106.0

### Notes:

You know that a service program makes its procedures available to external users through the export mechanism. The external users are modules and subprocedures in external programs and other service programs that call the procedures and subprocedures of the service program. These calls are import requests for the modules. The external users are also called *public* or *clients*.

A *signature* is a value that identifies the ILE *public interface* supported by a service program. It is similar in purpose to a level check on an IBM i file object. You may specify your own explicit signature, or the binder generates a signature based on the list of procedure and data item names to be exported and the sequence in which they are specified. Therefore, a signature provides a quick and convenient way to validate accessibility (public interface) to the service program. A signature does not validate the interface to a particular procedure within a service program.

The level check (**LVLCHK**) parameter on the **STRPGMEXP** command of the binder language specifies that, during the binding process, the public interface to the service

program should be checked. Specifying **LVLCHK(\*YES)**, or letting the value for **LVLCHK** default to **\*YES**, causes the binder to examine the signature parameter.

The signature is generated when the service program is created.

The signature of the service program is copied to every **\*PGM** object that binds by reference to that service program when the **\*PGM** is created. When the bind is completed at program load, the signatures are checked. If the signatures match, the clients referencing the service program can use the public interface specified by the signature without being recompiled.

As you have probably guessed, the signature can be used to make sure that the calling procedure is not referencing a **\*SRVPGM** that has been changed. In some cases, it may not be critical. Perhaps only logic within a procedure or subprocedure of the **\*SRVPGM** has changed. In situations that are not critical, the signature does not change. An example would be the situations where you have simply recreated the service program without changing the list of data or procedure items to be exported; in this case, the signature will not be changed.

In other cases, if a new parameter was added to a subprocedure within a **\*SRVPGM**, then the signature difference would present errors indicating that our source program had been changed.

Note that if the public interface does not change (even if you rebind), the signature does not change.

# Display service program's signature

IBM i

```

Display Service Program (DSPSRVPGM)

Type choices, press Enter.

Service program . . . . . NOMAIN
  Library . . . . . as10v6lib Name
  Output . . . . . * , *PRINT
Detail . . . . . *SIGNATURE *ALL, *BASIC, *SIZE...
+ for more values

Display Service Program Information Display 1 of 1
Service program . . . . . : NOMAIN
  Library . . . . . : AS10V6LIB
Owner . . . . . : EJJACKS
Service program attribute . . . . . : RPGLE
Detail . . . . . : *SIGNATURE

Signatures:
0000000000000000E2E8C1C4D9C2D5

```

Signature in character = SYADRBIN

© Copyright IBM Corporation 2011

Figure 6-27. Display service program's signature

AS106.0

## Notes:

You can display the signature of a service program by issuing the **DSPSRVPGM** command as shown above. The signature is normally displayed in hexadecimal, but it can be displayed as a character value. In this example, the binder generated the signature for the service program NOMAIN.

# The power of binder language and signatures

## (1 of 7)

IBM i

FILE: MYLIB/QSRVSRC

MEMBER: FINANCIAL

```
STRPGMEXP PGMLVL (*CURRENT)
  EXPORT SYMBOL ('Term')
  EXPORT SYMBOL ('Rate')
  EXPORT SYMBOL ('Amount')
  EXPORT SYMBOL ('Payment')

ENDPGMEXP
```

```
CRTSRVPGM SRVPGM (MYLIB/FINANCIAL)
  MODULE (MYLIB/MONEY MYLIB/RATES MYLIB/CALCS)
  EXPORT (*SRCFILE)
  SRCFILE (MYLIB/QSRVSRC)
  SRCMBR (*SRVPGM)
```

© Copyright IBM Corporation 2011

Figure 6-28. The power of binder language and signatures (1 of 7)

AS106.0

### Notes:

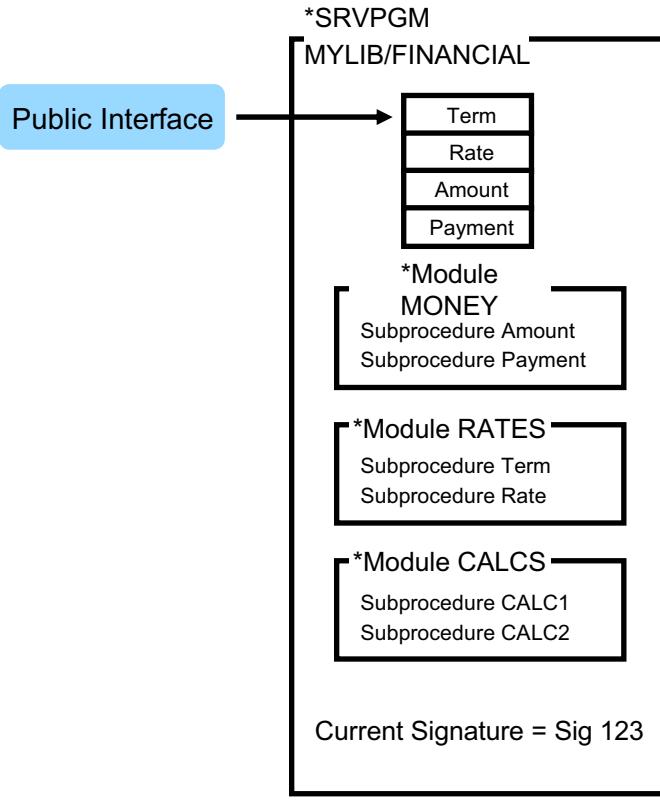
Assume that you are developing a simple financial application with the following subprocedures:

- **Rate** calculates an Interest\_Rate, given the values of Loan\_Amount, Term\_of\_Payment, and Payment\_Amount.
- **Amount** calculates the Loan\_Amount, given the values of Interest\_Rate, Term\_of\_Payment, and Payment\_Amount.
- **Payment** calculates the Payment\_Amount, given the values of Interest\_Rate, Term\_of\_Payment, and Loan\_Amount.
- **Term** calculates the Term\_of\_Payment, given the values of Interest\_Rate, Loan\_Amount, and Payment\_Amount.

These subprocedures are packaged in two modules named Money and Rates. A third module, Calcs, contains some generic subprocedures for the application. We could have specified a binding directory but instead coded the modules to be bound in the **CRTSRVPGM** command.

# The power of binder language and signatures (2 of 7)

IBM i



© Copyright IBM Corporation 2011

Figure 6-29. The power of binder language and signatures (2 of 7)

AS106.0

## Notes:

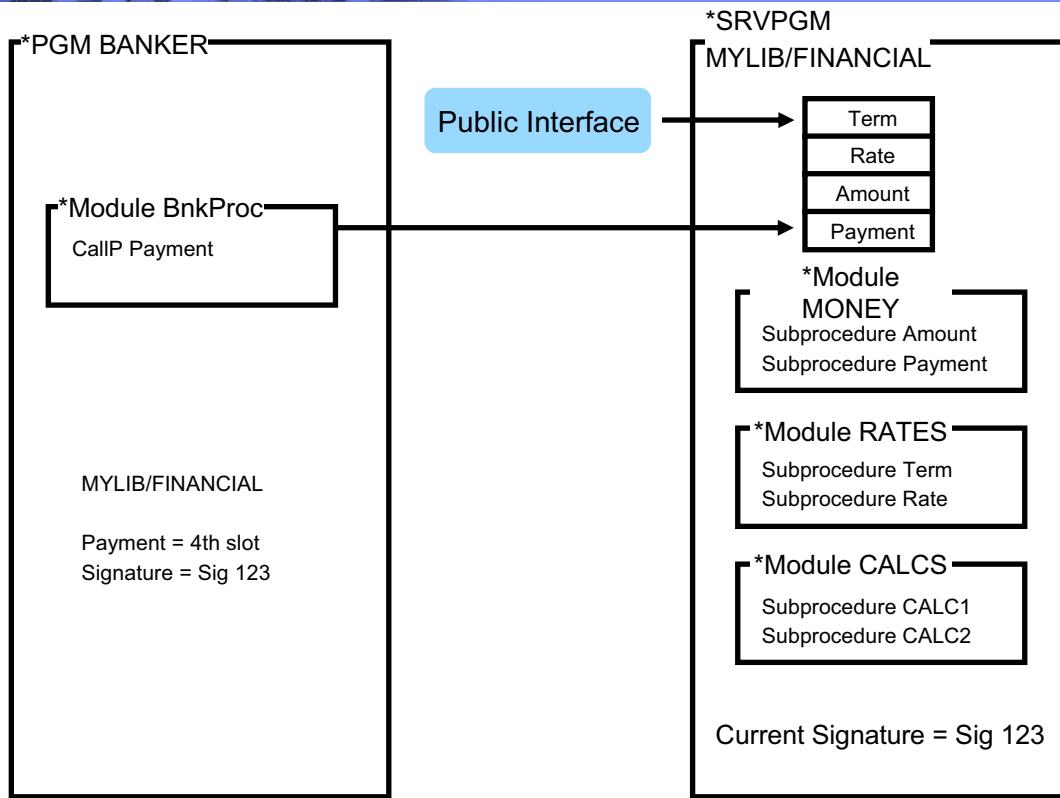
The public interface to the service program, Financial, is built via the exports and the signature. We have a signature of **123** for simplicity.

Pay attention to the order in which the exports are shown. This order is important later in the example as we show you an interesting feature of signatures and exports.

# The power of binder language and signatures

## (3 of 7)

IBM i



© Copyright IBM Corporation 2011

Figure 6-30. The power of binder language and signatures (3 of 7)

AS106.0

### Notes:

When we created the BANKER program some time ago, the MYLIB/FINANCIAL service program was provided on the BNDSRVPGM parameter. The symbol Payment was found to be exported from the fourth slot of the public interface of the FINANCIAL service program. The current signature of MYLIB/FINANCIAL, along with the slot associated with the Payment interface, is saved with the BANKER program.

When you call the BANKER program, activation verifies that:

- The service program FINANCIAL in library MYLIB exists.
- The service program still supports the signature (SIG 123) saved in BANKER.

This signature checking verifies that the public interface copied by BANKER when it was created is still valid at run time.

As shown in the visual, when BANKER is called, MYLIB/FINANCIAL still supports the public interface used by BANKER.

# The power of binder language and signatures (4 of 7)

IBM i

Enhance service program Financial:

1. Write the procedures OpenAccount and CloseAccount
2. FINANCIAL must support existing applications without rebinding!

Binder language (FILE: MYLIB/QSRVSRC MEMBER: FINANCIAL)

```
STRPGMEXP PGMLVL (*CURRENT)
  EXPORT SYMBOL('Term')
  EXPORT SYMBOL('Rate')
  EXPORT SYMBOL('Amount')
  EXPORT SYMBOL('Payment')
  EXPORT SYMBOL('OpenAccount')
    EXPORT SYMBOL('CloseAccount')
ENDPGMEXP
```

Supports  
New (\*Current)  
Applications

```
STRPGMEXP PGMLVL (*PRV)
  EXPORT SYMBOL('Term')
  EXPORT SYMBOL('Rate')
  EXPORT SYMBOL('Amount')
  EXPORT SYMBOL('Payment')
ENDPGMEXP
```

Supports  
Existing(\*Prv)  
Applications

Recreate Service Program FINANCIAL

```
CRTSRVPGM SRVPGM(MYLIB/FINANCIAL)
  MODULE(MYLIB/MONEY MYLIB/RATES MYLIB/CALCS MYLIB/ACCOUNTS))
  EXPORT(*SRCFILE)
  SRCFILE(MYLIB/QSRVSRC)
  SRCMBR(*SRVPGM)
```

3. Update the binder language to specify the new procedures

© Copyright IBM Corporation 2011

Figure 6-31. The power of binder language and signatures (4 of 7)

AS106.0

## Notes:

We decide to enhance the application, adding two new subprocedures. The two new subprocedures, OpenAccount and CloseAccount, open and close the accounts, respectively.

By using binder language, we can support the many programs that already call subprocedures in this service program by using the ability to support multiple signatures.

In this way, we avoid having to rebind existing programs until those programs need maintenance or enhancement.

# The power of binder language and signatures

## (5 of 7)

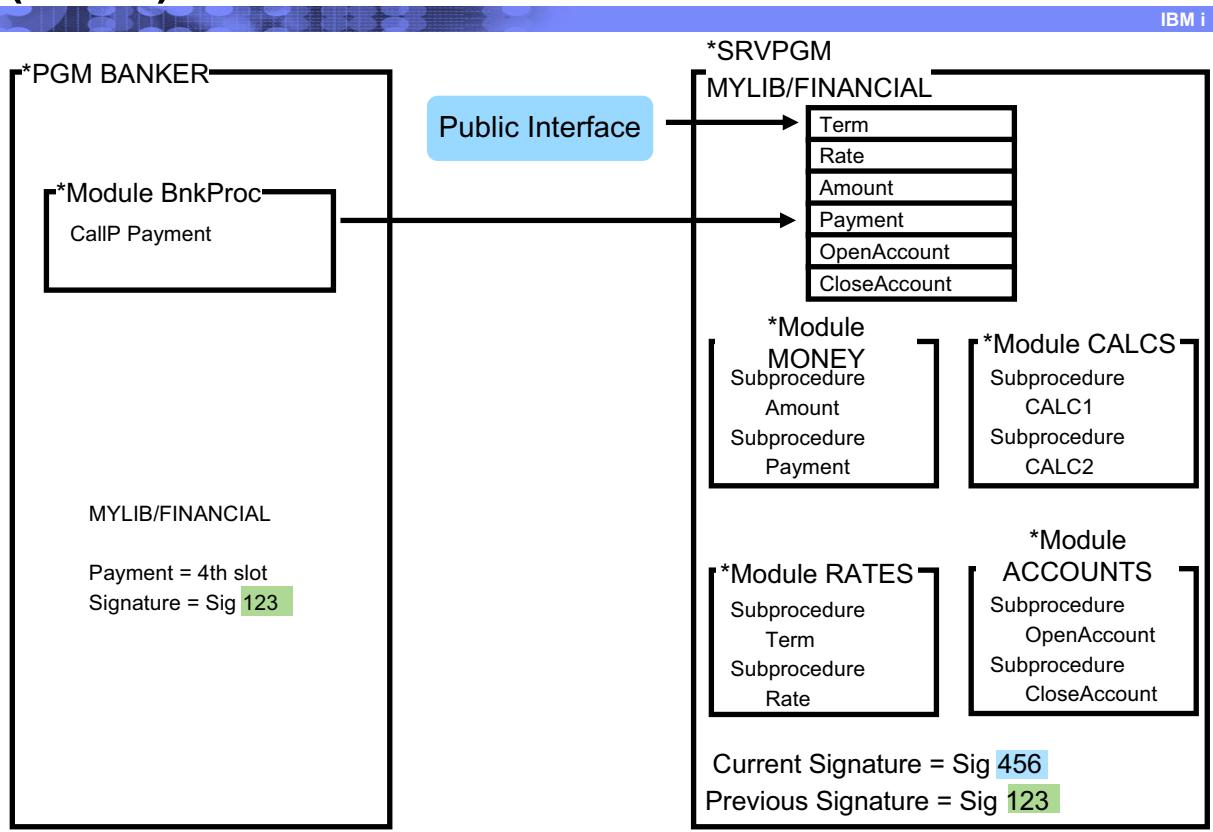


Figure 6-32. The power of binder language and signatures (5 of 7)

AS106.0

### Notes:

We did not need to rebind the BANKER program because the previous signature is still supported. Notice the previous signature in the service program MYLIB/FINANCIAL and the signature saved in BANKER. If BANKER were re-created by the CRTPGM command, the signature that is saved with BANKER would be the current signature of service program FINANCIAL.

The only reason we would ever need to re-create the program BANKER is if the program used one of the new subprocedures provided by the service program FINANCIAL.

The binder language allows you to enhance the service program *without changing existing programs and service programs* that use the changed service program.

# The power of binder language and signatures (6 of 7)

IBM i

1. Enhance the Rate subprocedure

2. Modify the binder language

Binder language (FILE: MYLIB/QSRVSRM MEMBER: FINANCIAL)

```

STRPGMEXP PGMLVL (*CURRENT)
  EXPORT SYMBOL('Term')
  EXPORT SYMBOL('Old_Rate') /* Original Rate procedure with four parameters */
  EXPORT SYMBOL('Amount')
  EXPORT SYMBOL('Payment')
  EXPORT SYMBOL('OpenAccount')
  EXPORT SYMBOL('CloseAccount')
  EXPORT SYMBOL('Rate') /* New Rate procedure that supports + a fifth parameter, Credit_History */
ENDPGMEXP

STRPGMEXP PGMLVL (*PRV)
  EXPORT SYMBOL('Term')
  EXPORT SYMBOL('Rate')
  EXPORT SYMBOL('Amount')
  EXPORT SYMBOL('Payment')
  EXPORT SYMBOL('OpenAccount')
  EXPORT SYMBOL('CloseAccount')
ENDPGMEXP

STRPGMEXP PGMLVL (*PRV)
  EXPORT SYMBOL('Term')
  EXPORT SYMBOL('Rate')
  EXPORT SYMBOL('Amount')
  EXPORT SYMBOL('Payment')
ENDPGMEXP

```

3. Recreate the service program Financial:

```

CRTSRVPGM SRVPGM(MYLIB/FINANCIAL)
  MODULE(MYLIB/MONEY MYLIB/RATES MYLIB/CALCS MYLIB/ACCOUNTS)
  EXPORT(*SRCFILE)
  SRCFILE(MYLIB/QSRVSRM)
  SRCMBR(*SRVPGM)

```

4. Must avoid rebinding existing programs

© Copyright IBM Corporation 2011

Figure 6-33. The power of binder language and signatures (6 of 7)

AS106.0

## Notes:

A fifth parameter, **Credit\_History**, is added on the call to the Rate subprocedure. **Credit\_History** updates the **Interest\_Rate** parameter that is returned from the Rate subprocedure.

The binder language in the visual supports the updated Rate subprocedure. It still allows existing ILE programs or service programs that use the FINANCIAL service program to remain unchanged.

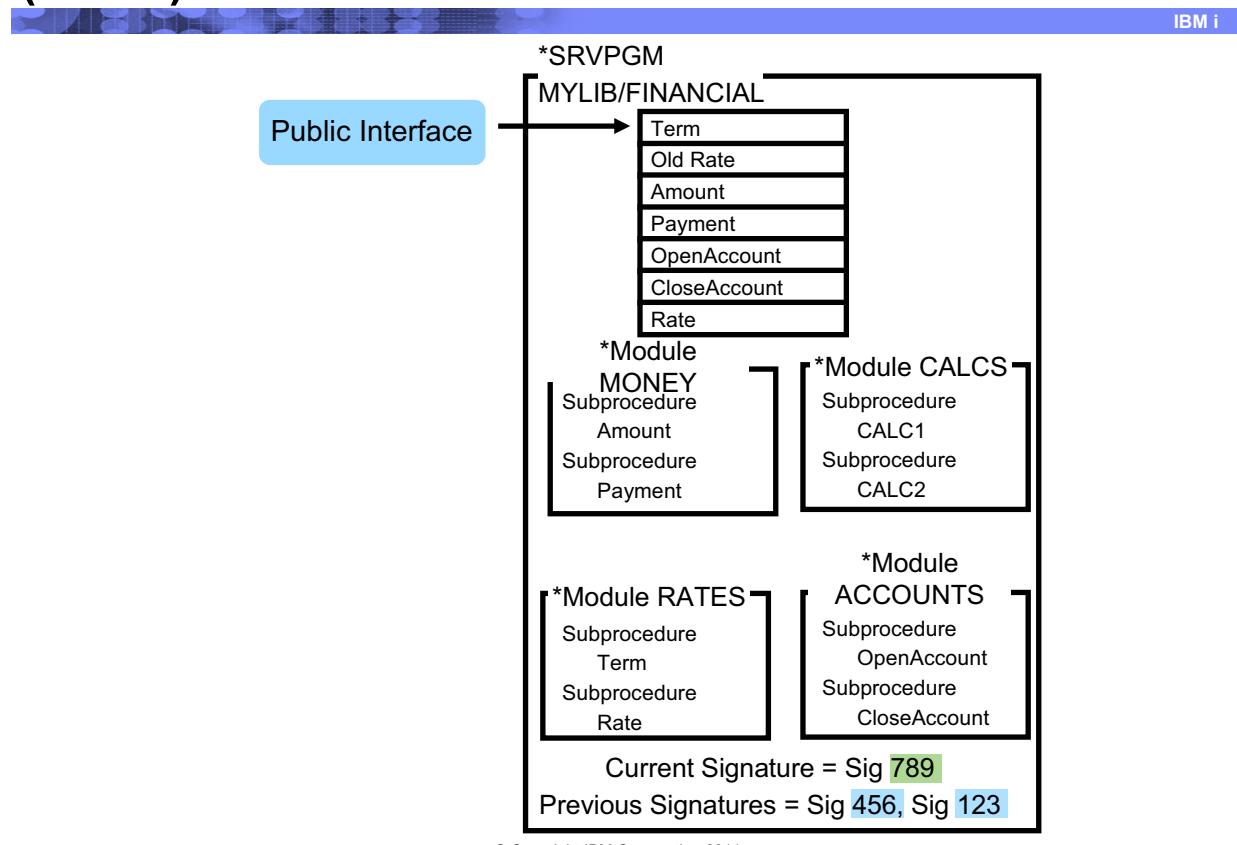
The original symbol Rate was renamed Old\_Rate but remains in the same relative position of symbols to be exported. *This is important to remember.*

A comment is associated with the Old\_Rate symbol. A comment is everything between /\* and \*/. The binder ignores comments in the binder language source when creating a service program.

The new subprocedure Rate, which supports the additional parameter of **Credit\_History**, must also be exported. This updated procedure is added to the end of the list of exports.

# The power of binder language and signatures

## (7 of 7)



© Copyright IBM Corporation 2011

Figure 6-34. The power of binder language and signatures (7 of 7)

AS106.0

### Notes:

Using the updated binder language and a new RATES module that supports the subprocedures Rate, Term, and Old\_Rate, we re-create the FINANCIAL service program shown above.

The ILE programs and service programs that reference the original subprocedure of the FINANCIAL service program go to slot 2 (in the visual) of the public interface. This directs the call to the Old\_Rate subprocedure, which is advantageous because Old\_Rate handles the original four parameters. Notice that we are taking advantage of the position of exports in the binder language to accomplish this.

If any of the ILE programs or service programs that used the original Rate subprocedure need to be re-created at some point in the future, do one of the following:

- To continue to use the original four-parameter Rate subprocedure, call the Old\_Rate subprocedure instead of the Rate subprocedure.
- To use the new Rate subprocedure, add the fifth parameter, **Credit\_History**, to each call to the Rate subprocedure.

# Managing multiple signatures

IBM i

## Good Practice:

```
STRPGMEXP PGMLVL (*CURRENT) LVLCHK (*YES)
SIGNATURE ('My_Sig_2')
```

```
STRPGMEXP PGMLVL (*PRV) LVLCHK (*YES)
SIGNATURE ('My_Sig_1')
```

## Avoid:

```
STRPGMEXP SIGNATURE (*GEN)
```

© Copyright IBM Corporation 2011

Figure 6-35. Managing multiple signatures

AS106.0

## Notes:

The RPG IV community of experts recommends that when managing multiple (compatible) signatures with PGMLVL(\*PRV), the programmer should explicitly control the signature:

```
STRPGMEXP PGMLVL (*CURRENT) LVLCHK (*YES) SIGNATURE ('My_Sig_2')
STRPGMEXP PGMLVL (*PRV) LVLCHK (*YES) SIGNATURE ('My_Sig_1')
```

Doing this has the benefit that the signature itself can be a meaningful string. One suggestion is to incorporate the \*SRVPGM name in the signature for readability when looking at DSPPGM results.)

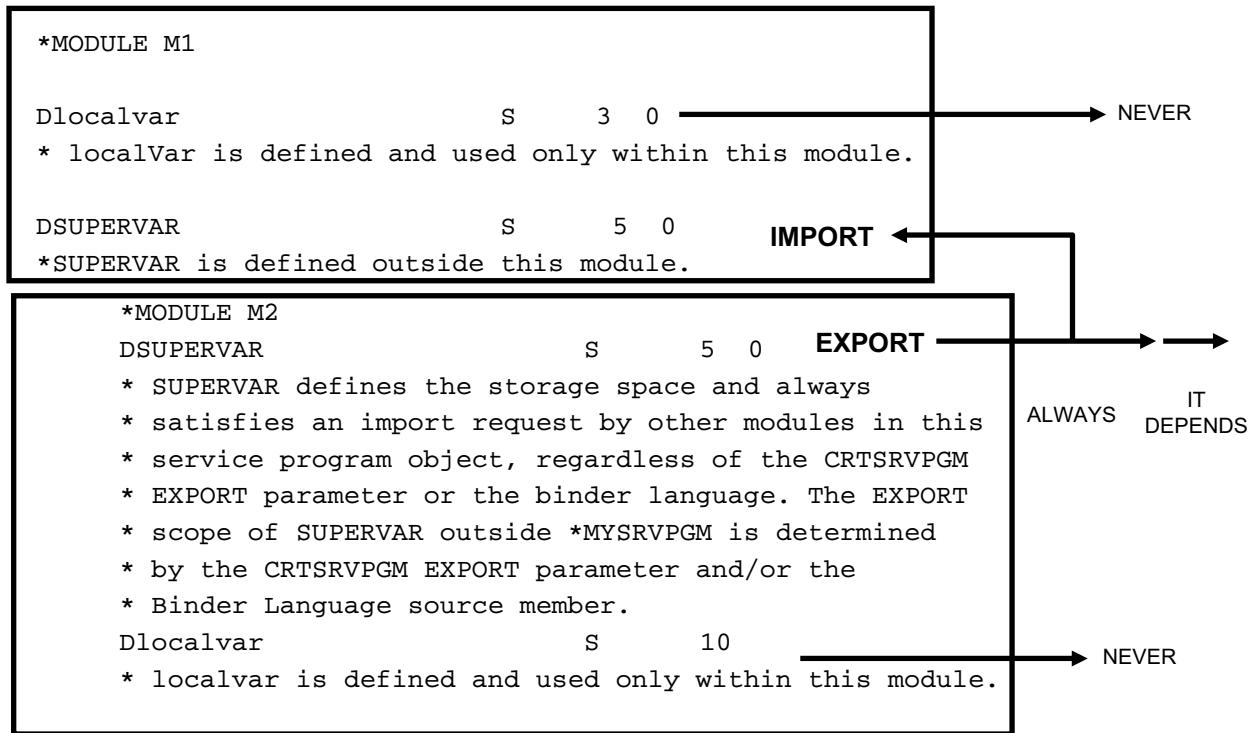
This means the signature changes only when appropriate. It can simplify matters considerably, but does introduce the potential for error. Under this regime, the rule is *always add new exports to the end of the list.*

The main reason for taking this approach is so that the application can be managed better through Version (Change) Control tools (such as Turnover or Aldon Change Management System (ACMS)).

# Binder language and scope of data export

IBM i

\*MYSRVPGM



© Copyright IBM Corporation 2011

Figure 6-36. Binder language and scope of data export

AS106.0

## Notes:

The **EXPORT** parameter of the **CRTSRVPGM** command specifies the names of the data and procedures this service program exports.

The possible values are:

**\*SRCFILE:** The source file and member specify the data and procedures to export from the service program.

**\*ALL:** All data and procedures that are exported from the specified modules are also exported from the service program.

If you code:

```
CRTSRVPGM MYSRVPGM EXPORT(*SRCFILE)
```

The binder language source member determines whether the data item, SUPERVAR, is exported from MYSRVPGM and is able to satisfy import requests from other modules.

However, if you code:

```
CRTSRVPGM MYSRVPGM EXPORT(*ALL)
```

SUPERVAR is exported from MYSRVPGM and is made available to satisfy import requests from other objects.



## 7.3. Binding directories

## What is a binding directory?

IBM i

- List of modules and service programs to be bound by **CRTPGM** or **UPDPGM**
- Holds a list of modules to be contained in service program using **CRTSRVPGM** or **UPDSRVPGM**
- Created and maintained using CL commands

© Copyright IBM Corporation 2011

Figure 6-37. What is a binding directory?

AS106.0

### Notes:

IBM i offers you several ways to make the process of specifying which modules to bind and which procedures and data items are to be exported and made available to other procedures not contained in this specific service program.

A *binding directory* simplifies the process of having to list the modules that are to be bound when you run the **CRTPGM** or **UPDPGM** commands. It may also be used to list the modules to be packaged in a service program using the **CRTSRVPGM** or **UPDSRVPGM** commands.

# Binding directory

IBM i

- Created and maintained using CL commands
- Create objects of type \*BNDDIR
- Binder matches import requests

*BNDDIR		
Object Name	Object Type	Object Lib
ALLOC	*SRVPGM	*LIBL
MATH	*SRVPGM	QSYS
FREE	*MODULE	*LIBL
HFREE	*SRVPGM	ABC

## CRTBNDDIR ADDBNDDIRE

© Copyright IBM Corporation 2011

Figure 6-38. Binding directory

AS106.0

### Notes:

A binding directory can be used instead of supplying a list of modules and service programs in your **CRTPGM** or **CRTSRVPGM** commands. A binding directory can also be used in addition to the list of modules specified in order to satisfy import requests. A binding directory consists of a list of modules and service programs that are candidates for *automatic binding*.

Not all binding entries in the list are necessarily bound. Only those required to satisfy exports that cannot otherwise be resolved are bound.

You might use a binding directory to contain modules and service programs that are standard procedures for your enterprise.

The most straightforward way to use a binding directory is to create one entry for each \*PGM or \*SRVPGM. The appropriate binding directory can be referenced as a parameter whenever you want to update either type of object. A binding directory reduces the number of parameters you must enter when creating a \*PGM or \*SRVPGM. Also, using a binding directory eliminates the possibility of making an error such as keying a wrong \*MODULE name.

Simply stated, a binding directory is just a search list used at bind time (**CRTPGM/CRTSRVPGM**) to resolve external references.

The visual shows the Create a Binding Directory (**CRTBNDDIR**) and Add a Binding Directory Entry (**ADDBNDDIRE**) commands. You can use these commands or the **WRKBNDDIRE** command to work with binding directory entries.

For the module (**MODULE**) parameter on the **CRTPGM** and **CRTSRVPGM** commands, there is a limit (of 150 parameters) on the number of modules you can specify. If the number of modules you want to bind exceeds the limit, a binding directory will avoid a problem.

# Using a binding directory

IBM i

- **WRKBNDDIR** provides access to all these commands:
  - **CRTBNDDIR**
  - **CRTBNDDIR BNDDIR(AS10V2LIB/AS10BNDDIR)**
  - **WRKBNDDIRE**
  - **ADDBNDDIRE**
  - **RMVBNDDIRE**
  - **DSPBNDDIR**
  - **DLTBNDDIR**
- Or use individual commands

© Copyright IBM Corporation 2011

Figure 6-39. Using a binding directory

AS106.0

## Notes:

This visual lists the commands that can be used to operate on binding directories. The main one is the **WRKBNDDIR** command, which allows access to all the others.

# WRKBNDDIR

IBM i

## Work with Binding Directories

Type options, press Enter.

1=Create 4=Delete 5=Display 9=Work with binding directory entries  
13=Change description

Opt	Binding Directory	Library	Text
<u>9</u>	<u>AS10BNDDIR</u>	<u>AS10V6LIB</u>	

Bottom

Parameters for options 1, 5 and 13 or command

====>

F3=Exit F4=Prompt F5=Refresh F9=Retrieve F11=Display names only  
F12=Cancel F16=Repeat position to F17=Position to

© Copyright IBM Corporation 2011

Figure 6-40. WRKBNDDIR

AS106.0

### Notes:

The Work with Binding Directory (**WRKBNDDIR**) command allows you to display and work with a list of binding directories. From this display, you can select from the Binding Directories to which you have \*USE authority.

In our example, we have created a single binding directory to which we add entries.

# WRKBNDDIRE

IBM i

Work with Binding Directory Entries

Binding Directory: AS10BNDDIR      Library: AS10V6LIB

Type options, press Enter.  
1=Add    4=Remove

Opt	Object	Type	Library	Activation	Date	Time	-----Creation-----
<u>1</u>	<u>nomain</u>	<u>*srvpgm</u>	<u>as10v6lib</u>				

(No binding directory entries for this binding directory.)

Bottom

Parameters or command  
====> \_\_\_\_\_

F3=Exit    F4=Prompt    F9=Retrieve    F5=Refresh    F12=Cancel    F17=Top  
F18=Bottom

© Copyright IBM Corporation 2011

Figure 6-41. WRKBNDDIRE

AS106.0

## Notes:

From the WRKBNDDIR display, option 9 brings you to this display, where you can add a list of modules or service programs to a binding directory one entry at a time.

# Binding directory entry added

IBM i

Work with Binding Directory Entries

Binding Directory: AS10BNDDIR Library: AS10V6LIB

Type options, press Enter.  
1=Add 4=Remove

Opt	Object	Type	Library	Activation	-----Creation-----	
					Date	Time
-	NOMAIN	*SRVPGM	AS10V6LIB	*IMMED	07/29/11	04:36:45

Bottom

Parameters or command  
====> \_\_\_\_\_

F3=Exit F4=Prompt F9=Retrieve F5=Refresh F12=Cancel F17=Top  
F18=Bottom

CRTPGM PGM(AS10V6LIB/AGEDEMOMN) BNDDIR(AS10V6LIB/AS10BNDDIR)

© Copyright IBM Corporation 2011

Figure 6-42. Binding directory entry added

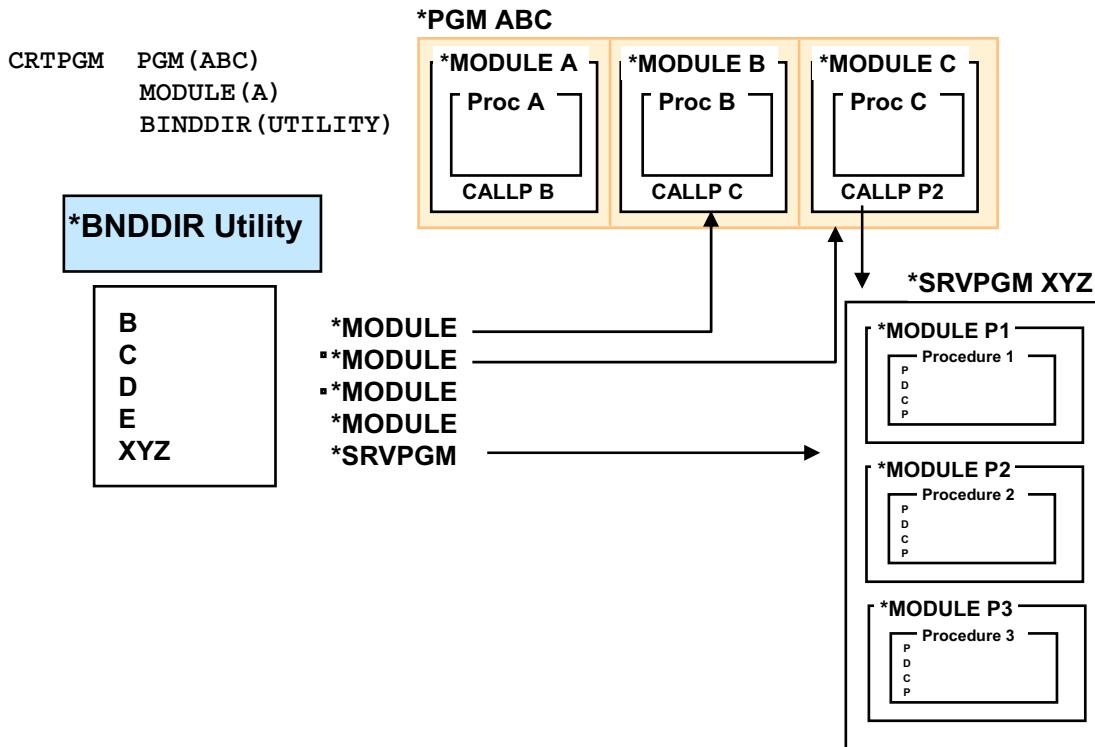
AS106.0

## Notes:

We add an entry to the binding directory. Now our \*SRVPGM NOMAIN is in the directory. We can add more entries as required. We can now bind the service program in a **CRTPGM** command or **CRTSRVPGM** command by specifying the binding directory AS10BNDDIR rather than the specific service program.

# Using a binding directory

IBM i



© Copyright IBM Corporation 2011

Figure 6-43. Using a binding directory

AS106.0

## Notes:

This visual is an example of creating a program using a binding directory. The **CRTPGM** command specifies bind by copy of module A plus a binding directory, UTILITY.

Notice that program ABC has import requests of modules A, B, and C. In addition, C has an import request of module P2, which is contained in service program XYZ.

In the visual, module B is bound to ABC because it satisfies an import specified by module A (CALLP B). Module C is bound to ABC because it satisfies an import specified by module B (CALLP C). Module P2 is bound (via \*SRVPGM XYZ) because it satisfies an import specified by module C (CALLP P2).

Remember that modules are bound only if they satisfy an import request. Additional modules that are contained in the service program XYZ (P1 and P3) are not imported and therefore are not bound.

In other words, an unreferenced module that is not specifically imported will not be bound to program ABC.

Names of service programs or modules in a binding directory can be specifically listed in the directory or they can be generic.

# Machine exercise: Using binding directories and binder language

IBM i



© Copyright IBM Corporation 2011

Figure 6-44. Machine exercise: Using binding directories and binder language

AS106.0

## Notes:

Perform the machine exercise *using binding directories and binder language*.



## 7.4. Activation groups

# Activation groups

IBM i

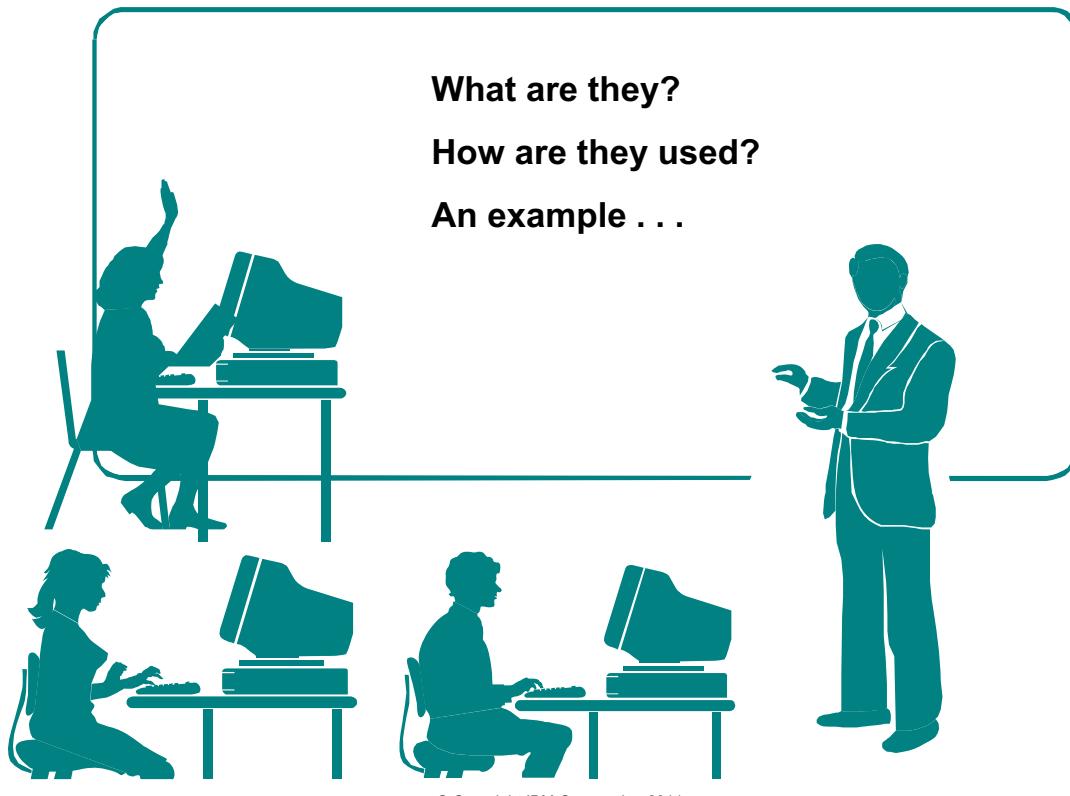


Figure 6-45. Activation groups

AS106.0

## Notes:

# What are activation groups?

IBM i

- An activation group is a division within a job
  - Each AG may own its own resources, such as:
    - File overrides
    - Shared file open data paths (ODPs)
    - Commitment control transactions
- ILE programs are created to run in a specific AG
  - Often an application boundary
  - `CRTPGM .... ACTGRP(name/*CALLER/*NEW/*ENTMOD)`

© Copyright IBM Corporation 2011

Figure 6-46. What are activation groups?

AS106.0

## Notes:

An *activation group* (also called an *AG*) is a division of a job. It can be thought of as a job within a job. Prior to ILE, certain types of a job's resources (mostly related to files) were considered to be sharable resources. That is, they are resources that are observed and used by more than one program.

Examples include overrides to files, shared open data paths (Share \*Yes), and commitment control transactions. However, prior to ILE, these resources can be shared among all programs in the job.

With activation groups, you can choose whether you want to share resources across the entire job (as you do in OPM) or only within an activation group, that is, among the programs running in the activation group. You can now have multiple commit transactions open in a job at the same time, for example, or you can have some overrides seen only by programs in one AG and not in another.

Some ILE compilers (for RPG and CL) have an option to generate programs that run in *OPM compatibility mode*, which allows you to use the new compiles to generate programs that can run in the default activation group and behave more like OPM programs.

# Using activation groups

IBM i

- AGs can protect resources for use by a specific application
  - Resources such as file overrides and shared open data paths
  - From interference by other applications
  - Packaged software can run more independently
  - Less dependence on the invocation stack
- AGs can be used to clean up application resources
  - Easily and specifically by application
  - May operate similar to a giant LR

© Copyright IBM Corporation 2011

Figure 6-47. Using activation groups

AS106.0

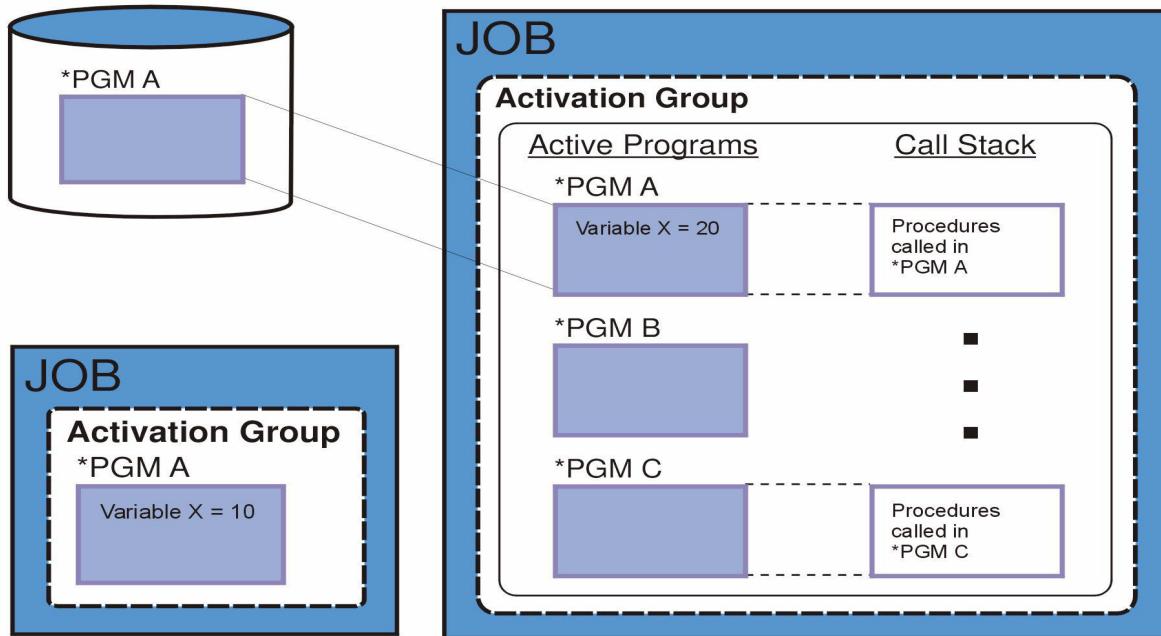
## Notes:

ILE activation groups are primarily used for one of these two purposes:

1. Since a given application can protect its own resources, such as file overrides and shared open data paths from other programs that may be running in the same job, it makes it easier to blend applications that use the same files without as much concern over what overrides or sharing options may be in effect.
2. Activation groups may also be used to clean up ILE program activation space used in the job. For OPM programs, this was normally accomplished with a **RCLRSC** command or a language-specific function, such as setting up last record (LR) indicator in RPG. ILE programs do not respond to those actions in the same way. To remove an ILE program activation from a job, one would typically reclaim the activation group in which the program is running in the job.

# What is activation?

IBM i



© Copyright IBM Corporation 2011

Figure 6-48. What is activation?

AS106.0

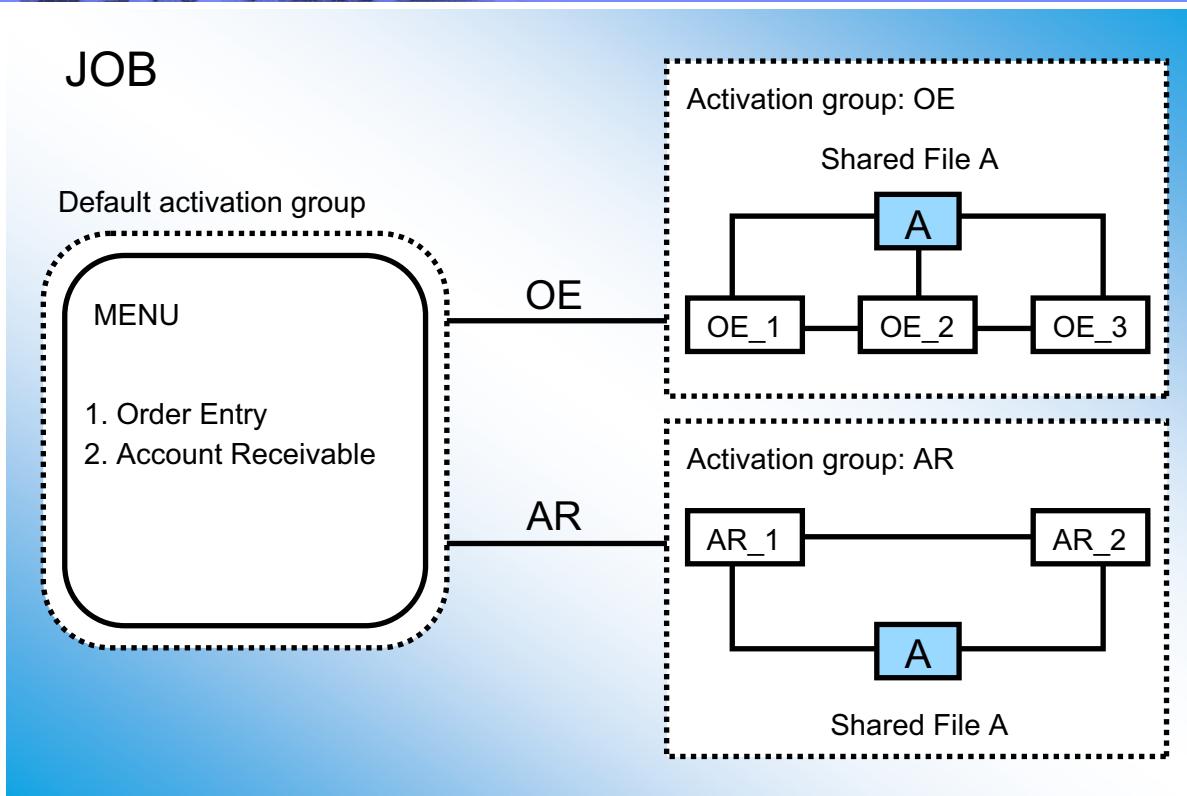
## Notes:

*Activation* is the process that is used to prepare a program to run. Both ILE programs and service programs must be activated by the system before they can be used within an IBM i job. Each activation is provided with one copy of static variables for each program activation. Only one copy of program instructions is provided across all activations.

All ILE programs and service programs are activated within an activation group. The activation group contains all the resources necessary to run the `*PGM` objects. We might think of an activation group as a subject, or a job within a job.

# Application isolation

IBM i



© Copyright IBM Corporation 2011

Figure 6-49. Application isolation

AS106.0

## Notes:

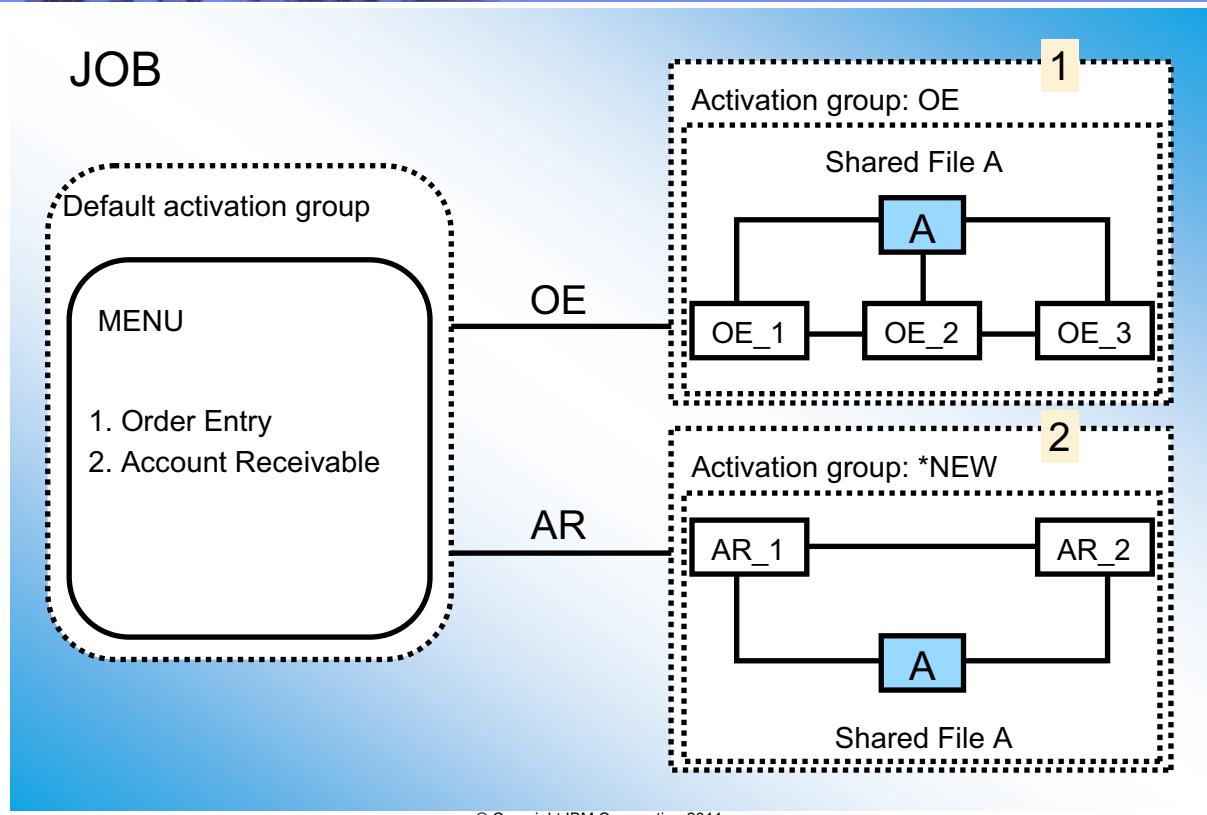
An activation group:

- Isolates applications within a job
  - Flexibility
  - Scoping of file overrides and commitment control
  - File sharing within activation groups
- Does selective cleanup of applications
- Uses firewalls (control boundaries to separate applications)
- Means that applications **own** resources
- Makes for less dependency on job call stack

The mechanism used to define the firewalls is the activation group. The developer can request that this application run in a unique activation group and that the resources, such as data file open data paths, be scoped to the activation group (as well as the job).

# Sample activation

IBM i



© Copyright IBM Corporation 2011

Figure 6-50. Sample activation

AS106.0

## Notes:

This visual illustrates a possible use of application-specific activation groups.

The order entry application has programs that are designated to run only in the OE activation group. The accounts receivable application has programs that are designated to run only in the AR activation group.

The applications are called from a menu which is written in an OPM language and is running in the default activation group.

**Note:** All OPM programs run in the default activation group. ILE programs typically run in ILE activation groups but have the ability to run in the default activation group as well.

File A is used by both applications. If a set of overrides are required for file A that *only* apply to the order entry activation group, a new parameter on the **OVRxxxF** command allows the programmer to specify those overrides apply only to programs in the OE AG using file A. Likewise, if the order entry application wants to share the open data path for file A, but does *not* want any other (non-order-entry) programs to use the same shared ODP, that can be specified. Commitment control can also be controlled by activation group.

Step by step, we need to allow the programs in the Accounts Receivable (AR) and Order Entry (OE) applications to run in their respective activation groups in order to use the features described in the previous visual.

1. Named ACTGRP(OE) is created when the first ILE program OE\_1 of the OE application is called (usually from a CLP using a CALLPRC).

This activation group remains active until:

- Hard leave: The ILE RPG language can issue a *hard leave* by issuing a bound CALLP to a bindable API that will delete the AG. The CEETREC API will perform a normal termination of the run unit and any pending changes for an ACTGRP level commitment definition. On the other hand, CEE4ABN causes abnormal termination of the run unit and any uncommitted changes for an ACTGRP level commitment definition are rolled back.
  - Reclaim activation group (**RCLACTGRP**)
  - Job end
2. ACTGRP(\*NEW) is deleted when both AR\_1 and AR\_2 are not on the call stack.
  3. OPM programs run in a special default activation group that is never deleted. The developer can specify the \*ACTGRP in which the program will run:

```
CRTPGM ..... ACTGRP | *NEW  
          | name  
          | *CALLER
```

```
CRTSRVPGM ... ACTGRP | *CALLER  
          | name
```

The programs in the visual were created using these commands:

```
CRTPGM PGM(OE_1) ... ACTGRP(OE)  
CRTPGM PGM(OE_2) ... ACTGRP(*CALLER)  
CRTPGM PGM(OE_3) ... ACTGRP(*CALLER)  
CRTPGM PGM(AR_1) ... ACTGRP(*NEW)  
CRTPGM PGM(AR_2) ... ACTGRP(*CALLER)
```

The visual shows the default activation group (contains the control language program (CLP)) and the OE and \*NEW activation groups.

1. Default (OPM) activation group is created at start of job.
2. The OE activation group created when OE\_1 is called. The OE AG is active until it is explicitly deleted or the job ends.
3. The AR activation group is created when AR\_1 is called. It is deleted when AR\_2 and AR\_1 are no longer on the call stack.

To fully exploit the power of user-defined activation groups, the developer can specify the activation group in which each ILE program is to run when the **CRTPGM** command is run.

**Note:** Be careful of using the default for **ACTGRP** parameter of **CRTPGM**. Be aware that starting in V5R3, when **ACTGRP(\*ENTMOD)** is specified, the program entry procedure module (**ENTMOD** parameter) is examined. If the module attribute is RPGLE, CBLLE, or CLLE, then **ACTGRP(QILE)** or **ACTGRP(QILETS)** is used. **QILE** is used when **STGMDL(\*SNGLVL)** is specified, and **QILETS** is used when System attribute that allows programs to use teraspace in their activation groups **STGMDL(\*TERASPACE)** is specified. If the module attribute is not ILE RPG program member type (RPGLE), ILE COBOL program member type (CBLLE), or ILE CL program member type (CLLE), then **ACTGRP(\*NEW)** is used. **\*ENTMOD** is the default value for the **ACTGRP** parameter.

To avoid any problems when you are creating program objects, you should always specify a specific activation group name, **\*NEW**, or **\*CALLER** and avoid **\*ENTMOD**.

# Creating activation groups

IBM i

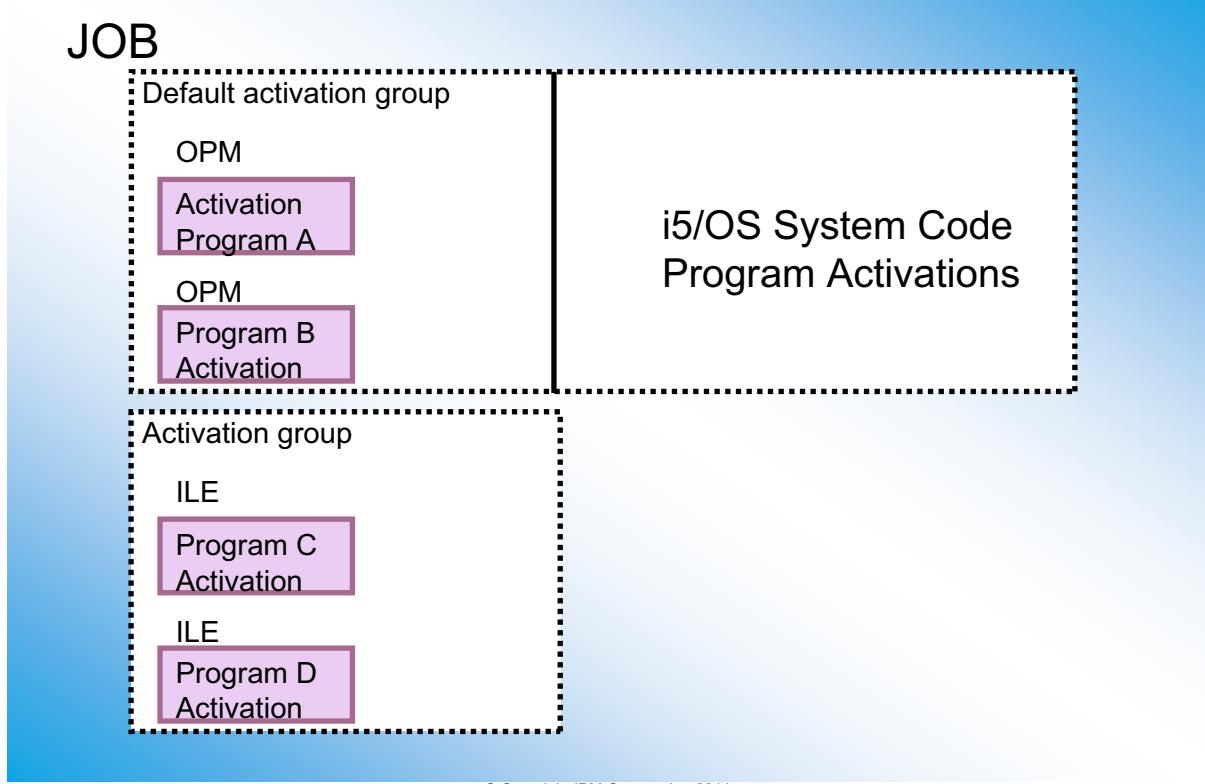


Figure 6-51. Creating activation groups

AS106.0

## Notes:

All ILE programs have one of the following activation group attributes:

- User-named activation group
- System-named activation group
- An attribute to use the activation group of the calling program

There are always *two* default activation groups. One is used by OPM programs, and the other one is used by IBM i system code.

You can control the creation of an ILE activation group by specifying an activation group attribute when you create your program or service program. Based on this attribute, an activation group is created by ILE as a part of dynamic program call processing.

This can be done either in **CRTPGM** or **CRTSRVPGM** by using the **ACTGRP** parameter, as follows:

- User-named activation group (permanent): **ACTGRP(name)**
- System-named activation group (temporary): **ACTGRP(\*NEW)**

- An attribute to use the activation group of the calling program: **ACTGRP(\*CALLER)**

All activation groups within a job have a name. Once an activation group exists within a job, it is used by ILE to activate programs and service programs that specify that name. You can not have duplicate activation group names in the same job.

### Default activation groups

When an IBM i job is started, the system creates two activation groups to be used by OPM programs. One activation group is reserved for IBM i system code. The other activation group is used for all other OPM programs. You cannot delete the OPM default activation groups. They are deleted by the system when your job ends.

Be aware of the system defaults for the activation group parameter for the **CRTPGM**, **CRTSRVPGM** and **CRTBNDRPG** commands because they are different and deserve some class discussion:

Command	Default	Valid Values
<b>CRTPGM</b> Activation group	*ENTMOD <sup>1</sup>	(Name, *NEW, *CALLER)
<b>CRTSRVPGM</b> activation group	*CALLER <sup>1</sup>	(Name, *CALLER)
<b>CRTBNDRPG</b> default activation group	*YES	(*YES, *NO)
<b>CRTBNDRPG</b> ILE activation group	QILE <sup>1,2</sup>	(Name, QILE, *NEW, *CALLER)

Notes:

- 1) Press F9 (All parameters) or F10 (Additional parameters) to see this.
- 2) Will also see this by changing **DFTACTGRP** parm to \*NO

It is important to understand the following activation group principles.

- **CRTPGM**: Every time you call a \*PGM that was created using **CRTPGM**, it will by default start in QILE (or QILETS) if the language of your entry module (\*ENTMOD) is CLLE, CBLLE or RPGLE. If your entry module is written in any other language, a new system-named activation group (\*NEW) will be used. This also allows a form of pseudo-recursion for RPG programs but is very inefficient. In many cases, it makes more sense to specify a named activation group for the job. For each job, the AG would be the same name.
- **CRTSRVPGM**: The default of \*CALLER makes sense since you would want the bind by reference to normally add the \*SRVPGM object to the AG from which it was called.

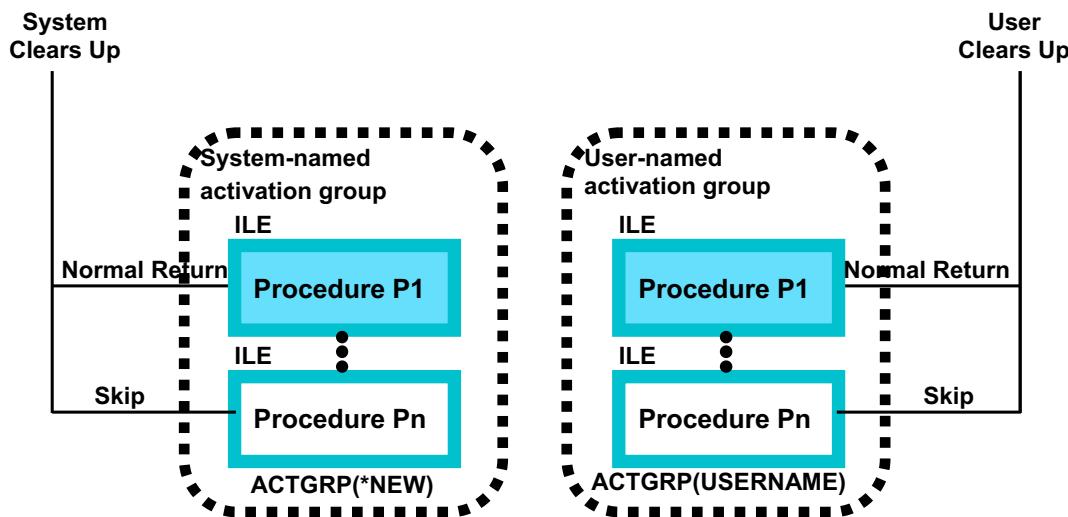
- **CRTBNDRPG:** This is important to understand. When you create an RPG program using **CRTBNDRPG**, you are creating a program with one module in it. The system assumes that it is an OPM RPG program. Therefore, you may only use a dynamic CALL to access other programs on the system. A bound CALLP is not valid.

You cannot use **CRTBNDRPG** when you want to create an ILE RPG \*PGM object if the program uses the bound CALLP opcode.

does

## Deleting an activation group

IBM i



© Copyright IBM Corporation 2011

Figure 6-52. Deleting an activation group

AS106.0

### Notes:

You can delete an activation group (non-DFTACTGRP) from your application job by using HLL end verbs.

An application may leave an activation group and return to a call stack entry that is running in another activation group in the following ways.

- Hard leave is caused by:
  - HLL end verbs (not available in RPG)
  - Unhandled exceptions such as leaving via a hard control boundary.

The first procedure called within a activation group is always the *hard control boundary* in that activation group.

Unhandled exceptions can be moved by the system to a call stack entry in another activation group.

- Skip operations: For example, sending an exception message or branching to a call stack entry that is not in your activation group.

- Soft leave caused by:
  - RPG RETURN opcode with LR on or off if ACTGRP(\*NEW)
  - CEETREC API
  - **RCLACTGRP** command for named ACTGRP, if not in use

# Activation group scoping

IBM i

- Some temporary resources may be scoped to AG
  - File open data paths
    - Protects shared ODP from other applications
    - **OPNSCOPE** parameter on **OPNDBF**, **OPNQRYF**, and **OVRxxxF** commands
  - File overrides
    - Protects other applications from overrides in AG
    - **OVRSCOPE** parameter on **OVRxxxF** commands
  - Commitment definitions
    - Protects inadvertent commit between applications
    - **CMTSCOPE** parameter on **STRCMTCTL** command

© Copyright IBM Corporation 2011

Figure 6-53. Activation group scoping

AS106.0

## Notes:

Other lesser used job resources may also be scoped to an AG:

- Local SQL cursors
- Remote SQL connections
- Hierarchical file system API
- User Interface Manager API
- Query management instances API
- Communications conversations

In OPM, you can share the ODP for files between programs in the same job. This can be done by specifying the **SHARE(\*YES)** parameter on the **CRTPF**, **CRTLTF**, **CHGPF**, **CHGLF**, or **OVRDBF** commands.

Using **SHARE(\*YES)** allows more than one program in the same job to share the file status, cursor position, and file storage area, and can improve performance by reducing the

amount of main storage the job needs and by reducing the time it takes to open and close the file.

# Types of scoping

IBM i

- There are three types of scoping possible:
  - Call-level scoping
    - Traditional OPM-type scoping
    - Overrides shared ODPs, for example, are used based on program's position in call stack
  - Activation group scoping
    - The default for programs running in ILE activation groups
    - AG owns the resource
    - Any shared resources available to all programs in AG regardless of call stack position
  - Job-level scoping
    - Available to both OPM and ILE programs
    - Resources available throughout the job
    - Must be explicitly requested, never the default

© Copyright IBM Corporation 2011

Figure 6-54. Types of scoping

AS106.0

## Notes:

*Call-level scoping* is what you have been using in OPM programs.

Looking at data files, in OPM, using the **SHARE(\*YES)** parameter lets two or more programs running in the same job share an open data path (ODP).

In ILE, shared files are scoped either to the activation group level or to the job level:

- **Activation group-level scoping** occurs when the data management resource is connected to the activation group of the ILE program or service program that created the resource. When the activation group is deleted, data management closes all resources associated with the activation group that have been left open by programs running in the activation group.

The ability to share a data management resource scoped to an activation group is limited to programs running in that activation group. This provides application isolation and protection, in the ILE environment.

- **Job-level scoping** occurs when the data management resource is connected to the job. Job-level scoping is available to both OPM and ILE programs. Job-level scoping

allows for sharing data management resources between programs running in different activation groups.

Job-level scoping allows the sharing of data management resources between all ILE and OPM programs running in the job.

OPNDBF OPNSCOPE (\*ACTGRPDFN/\*ACTGRP/\*JOB)

- **\*ACTGRPDFN**

The scope of the open operation is determined by the activation group of the program that called the **OVRDBF** command processing program. If the activation group is the default activation group, the scope is the call level of the caller. If the activation group is a non-default activation group, the scope is the activation group of the caller.

Note that this value is the default for the **OPNSCOPE** parameter. The **OPNSCOPE** parameter specifies the extent of influence (scope) of the open operation.

- **\*ACTGRP**

The scope of the open data path (ODP) is the activation group. Only shared opens from the same activation group can share this ODP. This ODP is not reclaimed until the activation group is deactivated or until the file is closed within the activation group.

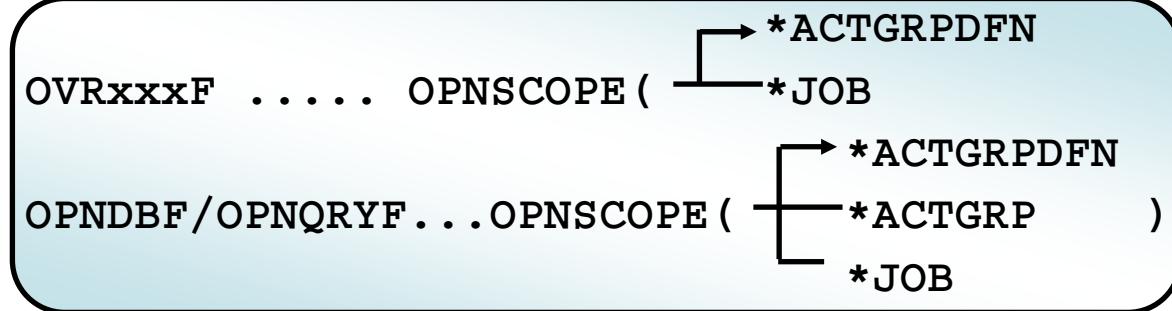
- **\*JOB**

The scope of the open operation is the job in which the open operation occurs.

# File open scoping

IBM i

- \*ACTGRPFDN
  - Call-level scoping if requested from default AG
  - Activation group scoping if requested from ILE AG
- \*JOB
  - Use job level scoping



© Copyright IBM Corporation 2011

Figure 6-55. File open scoping

AS106.0

## Notes:

The **OPNSCOPE** parameter on the override file commands allows you to specify the scope of the open data path for a file.

# File override scoping

IBM i

- \*ACTGRPFDN
  - Call-level scoping if requested from default AG
  - Activation group scoping if requested from ILE AG
- \*CALLVL
  - Call-level scoping for this override
- \*JOB
  - Use job-level scoping for this override



© Copyright IBM Corporation 2011

Figure 6-56. File override scoping

AS106.0

## Notes:

Overrides at the job level are processed last; that is, they override any other overrides.

In other words, if an override at activation group level says LPI=6 and one at job level says LPI=8, then LPI=8 will be in effect in that activation group.

Overrides can be merged, just as with call level overrides in OPM today.

## Further Detail

Overrides can be scoped to the:

- Activation group level: **\*ACTGRPFDN**

The scope of the override is determined by the activation group of the program that calls this command. When the activation group is the default activation group, the scope equals the call level of the calling program. When the activation group is not the default activation group, the scope equals the activation group of the calling program.

- Call-level: **\*CALLVL**

The scope of the override is determined by the current call level. All open operations done at a call level that is the same as or higher than the current call level are influenced by this override.

- Job level by specifying **OVRSCOPE(\*JOB) (\*CALLLVL)** on the **OVRDBF**

**OVRDBF OVRSCOPE (\*CALLLVL/\*JOB)**

The scope of the override is the job in which the override occurs.

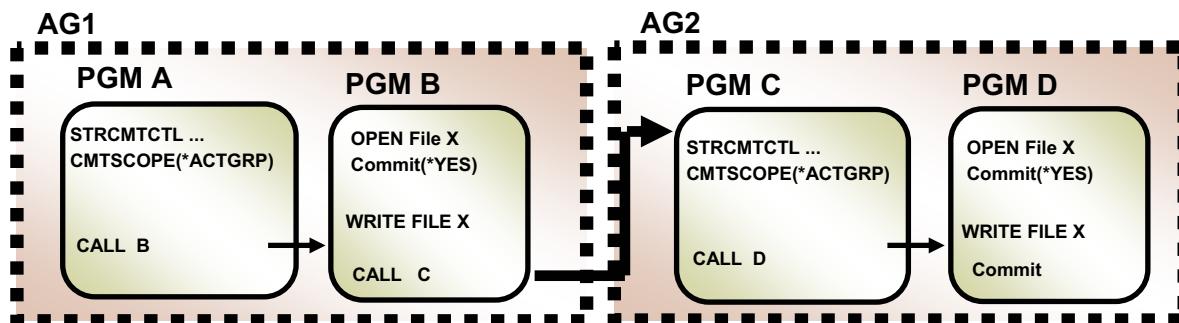
Using **OPNSCOPE** has no effect on sharing of ODPs unless the **SHARE(\*YES)** parameter is also specified on the open, the override, or on the file itself.

# Commitment control scoping

IBM i

- \*ACTGRP
  - Commitment definitions scoped to the AG
- \*JOB
  - Commitment definitions scoped to the job

```
STRCMTCTL .... CMTSCOPE ( [ *ACTGRP
                           *JOB ] )
```



© Copyright IBM Corporation 2011

Figure 6-57. Commitment control scoping

AS106.0

## Notes:

Multiple commitment definitions can be started and used by programs running within a job.

Each commitment definition for a job identifies a separate transaction that has associated with it committable resources that can be committed or rolled back independently of all other commitment definitions started for the job.

The scope for a commitment definition is used to indicate which programs that run within the job will use that commitment definition.

### STRCMTCTL

- \*ACTGRP
- \*JOB

In this example, the commit statement in program D commits *only* any records that are added, updated, or deleted in AG2.

The record added in AG1, in program B, is still uncommitted. Thus you can protect applications from one another in a commitment control environment.

## 7.5. Other ILE matters

## Recursive calls of subprocedures

IBM i

- Powerful feature
- Each call adds to call stack
- New storage for all data items
  - Data is unique to each invocation

© Copyright IBM Corporation 2011

Figure 6-58. Recursive calls of subprocedures

AS106.0

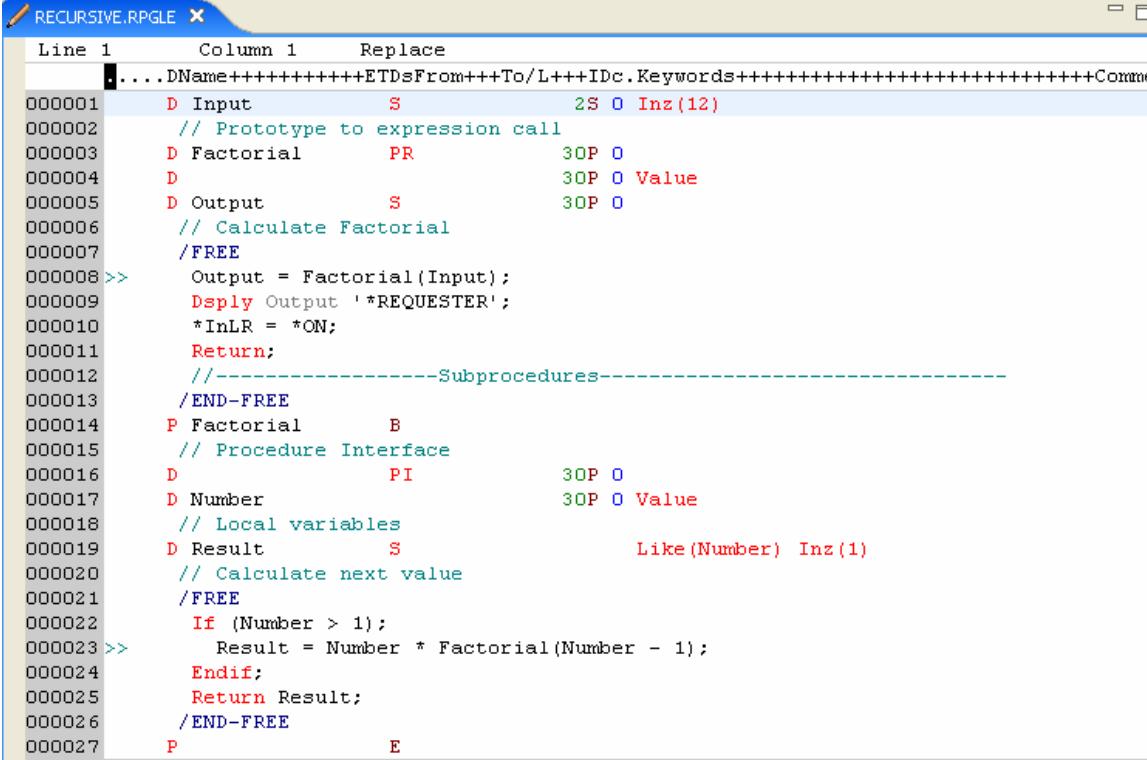
### Notes:

Subprocedures may be called recursively. A recursive call is one in which procedure A calls itself or calls B, which calls A again. For each CALLP, a new set of data items local to the subprocedure is allocated in automatic storage. Since each copy of these data items is LOCAL, they are hidden and unavailable to other calls. Some possible uses of recursive calls might be:

- When you are exploding a bill of materials you could recursively CALL a subprocedure as you explode the component parts down level by level.
- Iterative processing, such as route (delivery) planning and optimization. In this case, you could repeatedly call the subprocedure as you explored different route possibilities for the distribution of a product such as newspapers.

# Recursion: An example

IBM i



```

RECURSIVE.RPGLE X
Line 1      Column 1      Replace
. ....DName++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++Comme
000001      D Input          S           2S O Inz(12)
000002      // Prototype to expression call
000003      D Factorial       PR          30P O
000004      D                      30P O Value
000005      D Output          S           30P O
000006      // Calculate Factorial
000007      /FREE
000008 >>      Output = Factorial(Input);
000009      Dsplt Output '*REQUESTER';
000010      *InLR = *ON;
000011      Return;
000012      //-----Subprocedures-----
000013      /END-FREE
000014      P Factorial       B
000015      // Procedure Interface
000016      D             PI          30P O
000017      D Number          S           30P O Value
000018      // Local variables
000019      D Result          S           Like(Number) Inz(1)
000020      // Calculate next value
000021      /FREE
000022      If (Number > 1);
000023 >>      Result = Number * Factorial(Number - 1);
000024      Endif;
000025      Return Result;
000026      /END-FREE
000027      P             E

```

© Copyright IBM Corporation 2011

Figure 6-59. Recursion: An example

AS106.0

## Notes:

This visual shows you an example of calculating the factorial of a number passed to the subprocedure.

The subprocedure, Factorial, calls itself until the value of Number = 1.

Important note: the parameters should be passed by VALUE rather than the default, by reference, when using a recursive call.

This is an example of direct recursion, (A>A). You can also code an indirect recursion (A>B>A), which is often a cause of problems in an application.

## Encapsulating SQL based I/O in service programs

IBM i

- Isolates I/O from main procedures
- Uses embedded SQL to perform all I/O
- Package embedded SQL procedures as modules in service programs
- Can take advantage of functions and efficiency of SQL based I/O
  - Scoped delete, update
- Good implementation of modular programming: code reuse!
- Can organize I/O to minimize number of full opens/ODP creations

© Copyright IBM Corporation 2011

Figure 6-60. Encapsulating SQL based I/O In service programs

AS106.0

### Notes:

Now that you have a better understanding of the many features of ILE, let us explore another idea.

Some may not have used much SQL, but you should be aware that SQL offers a number of advantages over the legacy RPG IV I/O opcodes.

One of the features that many users of embedded SQL like is the ability to perform a group of updates or deletes in one statement rather than one record at a time. Of course, this feature of SQL must be carefully coded to make sure that you do not delete more records than you intended.

# Development process

IBM i

- Write embedded SQL module
  - Need PI/PR
  - Source type SQLRPGLE
- Write RPG IV program
  - Do not use I/O opcodes for data files
  - Code CALLP to embedded SQL module
  - Source type RPGLE
- Create modules
- Code binder language for exports from service program
- Create and add binding directory entries
- Create service program to contain embedded modules
  - Make activation group \*CALLER
- Create RPG IV main program that calls embedded SQL module

© Copyright IBM Corporation 2011

Figure 6-61. Development process

AS106.0

## Notes:

This visual summarizes the steps to follow when you separate I/O from a program.

# Using embedded SQL: Existing program example

IBM i

```

000001      FApInv_PF  IF   E          K Disk
000002      FEmbed01D  CF   E          Workstn InDDS(Indicators)
000003
000004      D  Indicators      DS
000005      D  Exit           1N  Overlay(Indicators:03)
000006      D  Cancel          1N  Overlay(Indicators:12)
000007      D  Error           1N  Overlay(Indicators:90)
000008
000009      /FREE
000010      Exfmt Prompt;
000011      Dow Not Exit;
000012      // SQL I/O Statements
000013
000014 >>    Exec SQL
000015 >>    Select VndNbr, ApInvNbr, ApCheck#
000016 >>    into :VndNbr, :ApInvNbr, :ApCheck#
000017 >>    from ApInv_FF
000018 >>    where PoNbr = :PoNbr;
000022
000023      // End SQL I/O Statements
000024      // Set Error indicator using SQLCOD
000025 >>    Error = (SqlCod <> 0);
000026
000027 >>    Dow SqlCod = 0;
000028      Exfmt Detail;
000029      Select;
000030      When Exit;
000031      *InLR = *ON;
000032      Return;
000033
000034      When Cancel;
000035      Leave;
000036      Ends1;
000037      Enddo;
000038
000039      Exfmt Prompt;
000040      Enddo;
000041
000042      *InLR = *ON;
000043      Return;
000044
000045      /END-FREE

```

© Copyright IBM Corporation 2011

Figure 6-62. Using embedded SQL: Existing program example

AS106.0

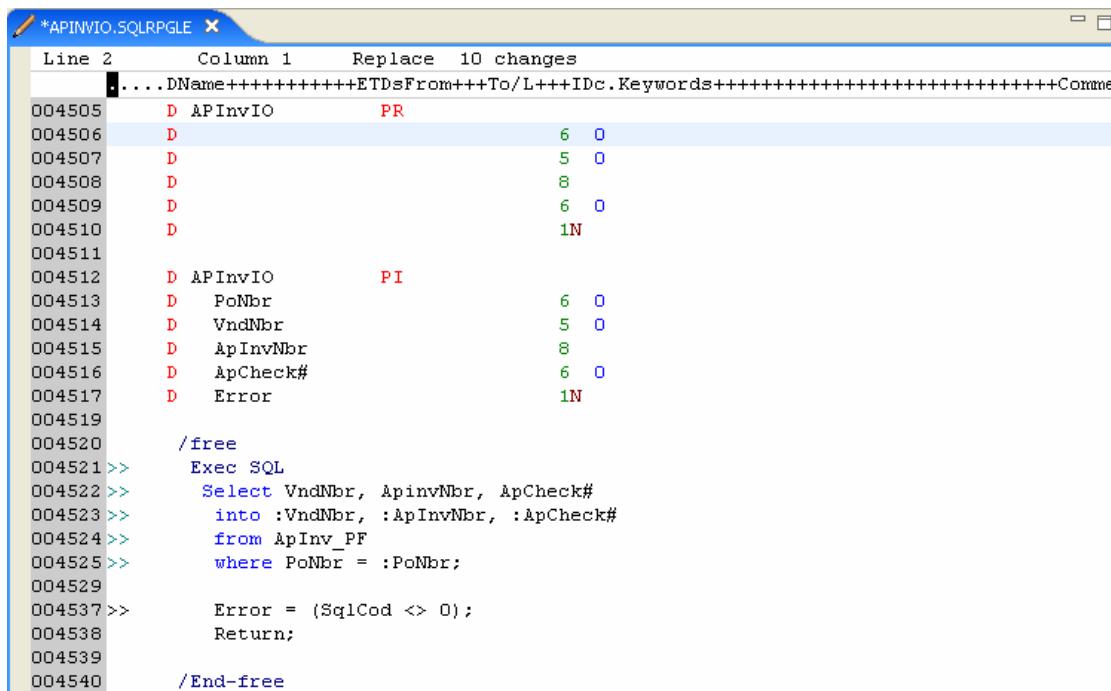
## Notes:

You may remember this program from an earlier unit. It uses embedded SQL (Select statement) to perform I/O rather than a CHAIN operation code.

Suppose we were to move the IO (Select) into a separate procedure. How would we do this?

# Using embedded SQL: I/O module

IBM i



```
*APINVIO.5QLRPGLE X
Line 2      Column 1    Replace 10 changes
. .... DName++++++ETDsFrom++To/L+++IDc.Keywords+++++++++++++++++Comme
004505     D APInvIO          PR
004506     D                      6  0
004507     D                      5  0
004508     D                      8
004509     D                      6  0
004510     D                      1N
004511
004512     D APInvIO          PI
004513     D PoNbr                  6  0
004514     D VndNbr                  5  0
004515     D ApInvNbr                8
004516     D ApCheck#                 6  0
004517     D Error                   1N
004519
004520     /free
004521 >>   Exec SQL
004522 >>     Select VndNbr, ApInvNbr, ApCheck#
004523 >>       into :VndNbr, :ApInvNbr, :ApCheck#
004524 >>       from ApInv_PF
004525 >>       where PoNbr = :PoNbr;
004529
004537 >>     Error = (SqlCod <> 0);
004538     Return;
004539
004540     /End-free
```

© Copyright IBM Corporation 2011

Figure 6-63. Using embedded SQL: I/O module

AS106.0

## Notes:

This procedure performs the I/O. Notice that there is no need to define the APINV\_PF file on an F-spec since SQL makes a direct call to the DB2 database manager.

We coded the PI and PR for this procedure. Notice that we set an error indicator based upon the SQLCOD and return **Error** as a parameter to the calling procedure.

# Using embedded SQL: Main procedure

IBM i

```

00200  FEmbedOID  CF   E          Workstn  InDDS(Indicators)
00300
00400  D  Indicators      DS
00500  D  Exit           1N  Overlay(Indicators:03)
00600  D  Cancel          1N  Overlay(Indicators:12)
00700  D  Error           1N  Overlay(Indicators:90)
00800
00801  D  APIInvIO       PR
00802  D                  6  0
00803  D                  5  0
00804  D                  8
00805  D                  6  0
00806  D                  1N
00807
00900  /FREE
01000  Exfmt Prompt;
01100  Dow Not Exit;
01200  // Call to I/O Module; note Error Indicator passed from module
01201>> CALLP APIInvIO (PoNbr : VndNbr : ApInvNbr : ApCheck# : Error);
002600
002700>>    Dow Error = *off;
002800    Exfmt Detail;
002900    Select;
003000    When Exit;
003100      *InLR = *ON;
003200      Return;
003300
003400      When Cancel;
003500      Leave;
003600      Endsl;
003700      Enddo;
003800
003900      Exfmt Prompt;
004000      Enddo;
004100
004200      *InLR = *ON;
004300      Return;
004400
004500  /END-FREE

```

© Copyright IBM Corporation 2011

Figure 6-64. Using embedded SQL: Main procedure

AS106.0

## Notes:

This visual shows you the procedure without the embedded SQL. In its place, we coded a PR and added a CALLP to the module that will perform the I/O.

# Using embedded SQL: Commands used to create application

IBM i

1. CRTSQLRPGI OBJ(APINVIO) COMMIT(\*NONE)
2. CRTRPGMOD MODULE(APINQSUB)
3. Code binder language source for member APPLIO service program):
 

```
STRPGMEXP PGMLVL(*CURRENT)
EXPORT SYMBOL(APINVIO)
ENDPGMEXP
```
4. CRTSRVPGM SRVPGM(AS10V6LIB/APPLIO)
 MODULE(AS10V6LIB/APINVIO)
5. CRTPGM PGM(AS10V6LIB/APINQ)
 MODULE(AS10V6LIB/APINQSUB)
6. BNDSRVPGM(AS10V6LIB/APPLIO) ACTGRP(\*CALLER)

© Copyright IBM Corporation 2011

Figure 6-65. Using embedded SQL: Commands used to create application

AS106.0

## Notes:

You can use this as a reference when you create modular ILE applications.

## Checkpoint (1 of 2)

 IBM i

1. True or False: An ILE module can be composed of one or more procedures or subprocedures.
  
2. In order to make data items available outside an individual module:
  - a. Binder language must be used.
  - b. Use the EXPORT keyword.
  - c. Bind the modules by copy.
  - d. Bind modules by reference.
  
3. True or False: An ILE program can be composed of one or many modules.

© Copyright IBM Corporation 2011

---

Figure 6-66. Checkpoint (1 of 2)

AS106.0

### Notes:

## Checkpoint (2 of 2)

IBM i

4. An ILE RPG module:
  - a. May contain a main procedure consisting of H, F, D, I, C, and O specifications.
  - b. May contain zero or more subprocedures coded with P, D, and C specifications.
  - c. May make subprocedures available outside of the module in which they are contained.
  - d. All of the above.
5. True or False: In order for an ILE RPG module to be created with only sub procedures, the NOMAIN keyword must be specified in an H-specification.
6. True or False: The EXPORT keyword is all that is needed to insure that a subprocedure is available outside of the module in which it is packaged.
7. A \_\_\_\_\_ simplifies the process of having to list the modules that are to be bound when you run the CRTPGM, UPDPGM, CRTSRVPGM, or UPDSRVPGM commands.
  - a. Module list
  - b. Binding list
  - c. Binding directory
  - d. Module directory

© Copyright IBM Corporation 2011

Figure 6-67. Checkpoint (2 of 2)

AS106.0

### Notes:

## Unit summary

IBM i

Having completed this unit, you should be able to:

- Create modules that contain more than one procedure or subprocedure
- Create NOMAIN modules to contain one or more subprocedures
- Explain the use of EXPORT for data and module sharing
- Create and use binding directories
- Use binder language for exporting data and modules
- Describe and use activation groups

© Copyright IBM Corporation 2011

Figure 6-68. Unit summary

AS106.0

### Notes:

# Unit 8. ILE error handling and condition handlers

## What this unit is about

This unit describes error message handling in ILE and the use of ILE condition handlers to trap and handle run-time errors.

We will contrast ILE error handling to original program model (OPM) error handling as well.

## What you should be able to do

After completing this unit, you should be able to:

- Describe the hierarchy of error handling in ILE
- Define and differentiate promote and percolate
- Describe how to use an ILE condition handler
- Code an RPG IV ILE condition handler

## How you will check your progress

Accountability:

- Machine exercise
- Checkpoint questions

## References

SC41-5606      *ILE Concepts - Version 7*

SC09-2507      *Rational Development Studio for i ILE RPG Programmer's Guide Version 7*

## Unit objectives

IBM i

After completing this unit, you should be able to:

- Describe the hierarchy of error handling in ILE
- Define and differentiate promote and percolate
- Describe how to use an ILE condition handler
- Code an RPG IV ILE condition handler

© Copyright IBM Corporation 2011

Figure 7-1. Unit objectives

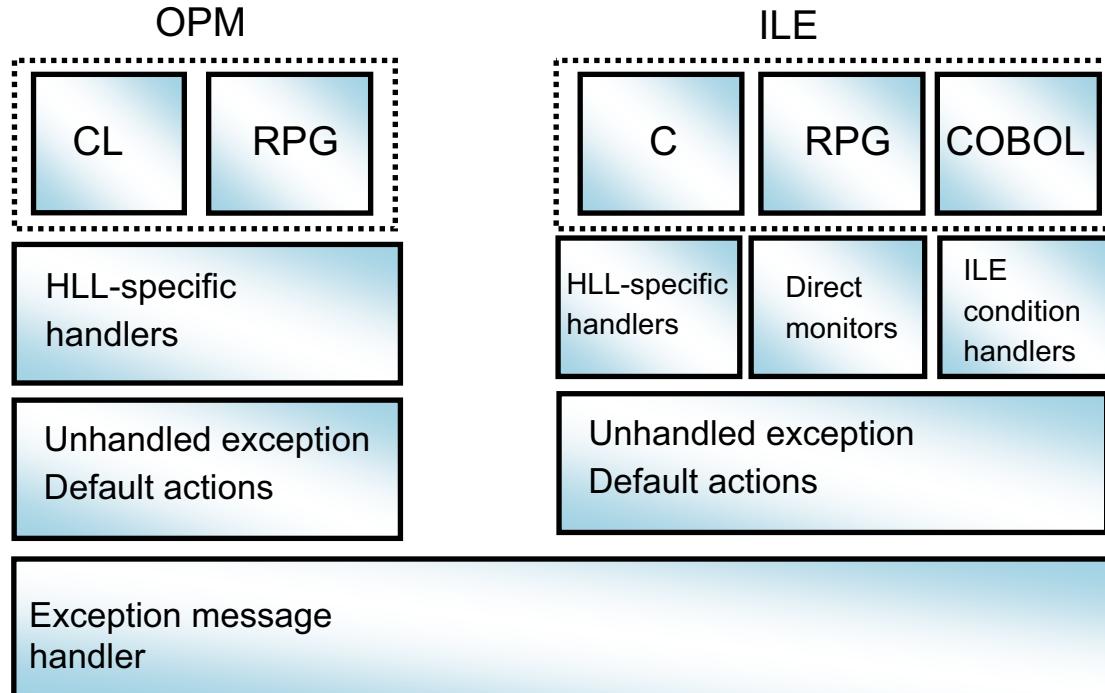
AS106.0

### Notes:

## 8.1. An overview of ILE error handling

# Error handling

IBM i



© Copyright IBM Corporation 2011

Figure 7-2. Error handling

AS106.0

## Notes:

These are the different layers in IBM i for error handling, and this visual shows the complete error-handling structure for both OPM and ILE programs.

ILE shares the same exception message architecture used by OPM programs. Exception messages generated by the system initiate language-specific error handling within an ILE program, just as they do within an OPM program. The lowest layer in the figure includes the capability for you to send and receive exception messages. This can be done with message handler APIs or commands. Exception messages can be sent and received between ILE and OPM programs.

In the case of OPM, (left side of visual), HLL-specific error handling provides one or more handling routines for each call stack entry. The appropriate routine is called by the system when an exception is sent to an OPM program.

An unhandled exception generates a special escape message known as a *function check message*. This message is given the special message ID of CPF9999. It is sent to the program message queue of the call stack entry that incurred the original exception message. If the function check message is not handled, the system removes that call stack

entry. The system then sends the function check message to the previous call stack entry. This process continues until the function check message is handled. If the function check message is never handled, the job ends.

HLL-specific error handling in ILE (right side of visual) provides the same capabilities. ILE however, has additional types of exception handlers. These types of handlers give you direct control of the exception message architecture and allow you to bypass HLL-specific error handling.

**Direct monitor handlers:** Allow you to directly declare an exception monitor around limited HLL source statements (available in RPG IV in V5R1 and above).

**ILE condition handlers:** Allow you to register an exception handler at run-time. ILE condition handlers are registered for a particular call stack entry.

**HLL-specific handlers:** Are the language features defined for handling errors in both OPM and ILE. HLL-specific error handling in RPG includes the ability to code \*PSSR and INFSR subroutines.

In ILE, an unhandled exception message is percolated to the previous call stack entry message queue. Percolation occurs when the exception message is moved to the previous call message queue. This creates the effect of sending the same exception message to the previous call message queue. When this happens, exception processing continues at the previous call stack.

## Types of error messages

IBM i

- ESCAPE: Severe error detected
- STATUS: Displays status of work done
- NOTIFY: Requires corrective action or reply
- Function check: You did not handle the exception

© Copyright IBM Corporation 2011

Figure 7-3. Types of error messages

AS106.0

### Notes:

Let us quickly review the types of messages you might get when an exception occurs while your program is executing.

When your program experiences a run-time error, an exception message is generated.

The exception message depends upon the severity of the error caused by your program:

- **\*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.

A *function check* is generated when one of the three previous exceptions occurs and is not handled by your program.

# Job message queues

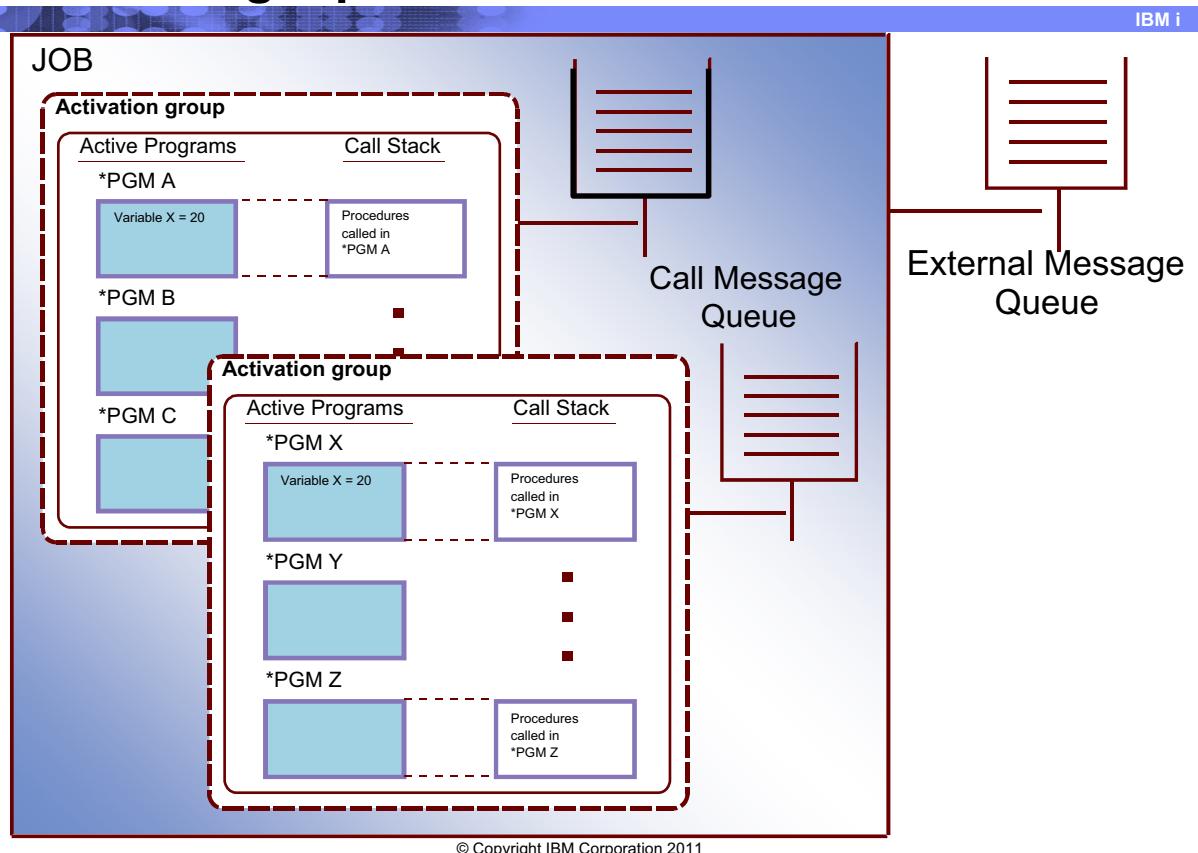


Figure 7-4. Job message queues

AS106.0

## Notes:

A message queue exists for every call stack entry within each IBM i job. This message queue facilitates the sending and receiving of informational messages and exception messages between the programs and procedures running on the call stack.

The call stack message queue is identified by the name of the OPM program or ILE procedure that is on the call stack. The procedure name or program name can be used to specify the target call stack entry for the message that you send. Because ILE procedure names may not be unique, the ILE module name and ILE program or service program name can optionally be specified. If you always qualify in this manner, then you will not have to be concerned about possible confusion. When the same program or procedure has multiple call stack entries, the nearest call message queue is used.

Each IBM i job has one external message queue. All programs and procedures running within the job can send and receive messages between an interactive job and the workstation user or QSYSOPR \*MSGQ by using this queue.

# Exception handlers in RPG IV

IBM i

- RPG IV language based handlers:
  - E opcode extender
    - Anticipates problem
    - Traps exception; continues execution
    - Replaces traditional error indicator
  - Monitor group
    - Can handle exceptions beyond scope of E opcode extender
    - For example, decimal data error
  - Error subroutines
    - Capture error information
    - Can assist in problem diagnosis
    - \*PSSR and INFSR
      - Can be used to percolate error
- ILE error handling facilities:
  - Condition handlers
    - User-written exception handler code
  - Cancel handlers
    - User-written code for abnormal termination

© Copyright IBM Corporation 2011

Figure 7-5. Exception handlers in RPG IV

AS106.0

## Notes:

ILE offers additional error handling options to the RPG IV programmer. ILE applications may use the traditional error-handling techniques, such as the E-extenders, monitor groups, and \*PSSR subroutines or INFSR subroutines. The E-extender is strongly recommended over the use of an error indicator. Your code is more easily understood during program maintenance. Using the E-extender enables you to anticipate a problem, trap it, and continue program execution.

Using the subroutine method for error handling while you are unable to continue execution, enables restart and debug information to be recorded to aid diagnostic work.

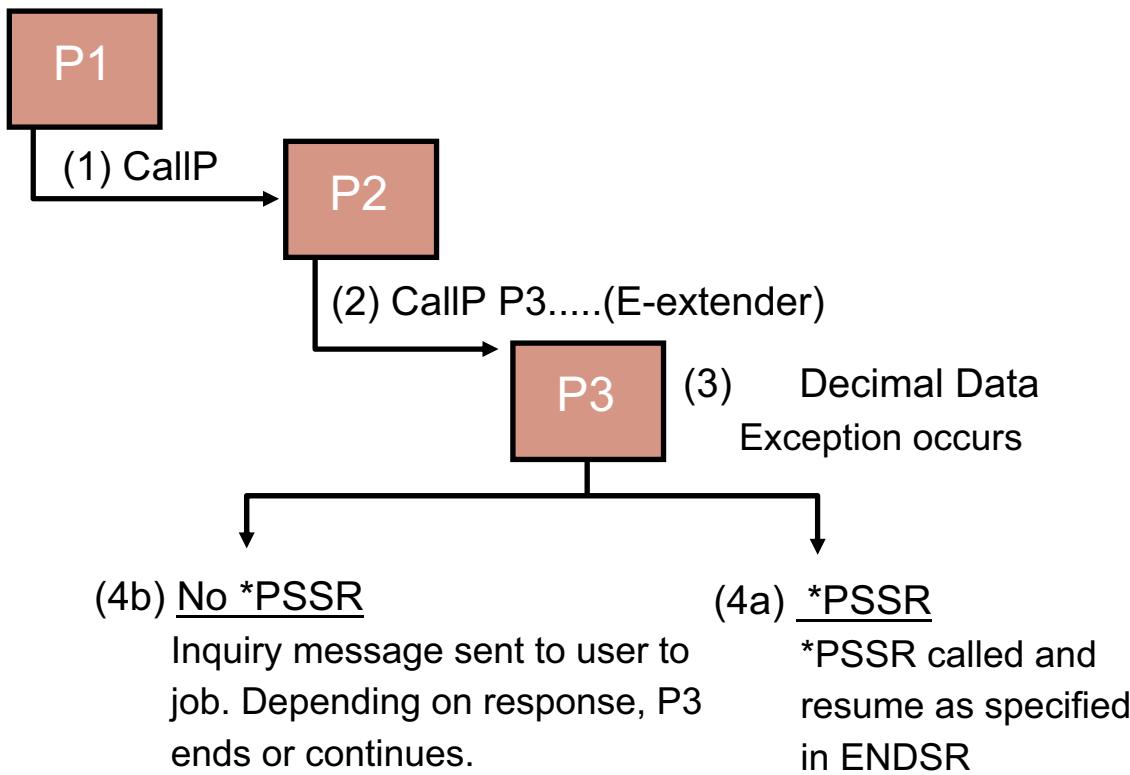
In addition to these, ILE offers:

1. User-written procedures that can handle exception conditions
2. User-written procedures that can handle abnormal end

We discuss these functions in this topic. There are some differences in how ILE applications behave when error conditions occur compared to traditional OPM applications. We also discuss these differences.

# OPM traditional exception handling

IBM i



© Copyright IBM Corporation 2011

Figure 7-6. OPM traditional exception handling

AS106.0

## Notes:

We use the example in the visual above to review how OPM error handling behaves. OPM means that all programs are running in the default activation group:

1. Program P1, running in the default activation group, calls program P2.
2. Program P2 calls program P3. Assume that program P2 has an E-extender (E-operation code extender in RPG IV) and %Error coded on the CALL statement and that there is no \*PSSR (Program Status Subroutine) error handling subroutine coded in P2.
3. Program P3 encounters a run-time decimal data error. No E-extender or %Error is coded on the statement where the decimal data error occurs.
4. (a) The \*PSSR is called and execution is resumed in a limited sense dependent upon the resume point specified in the ENDSR (operation code that ends a subroutine) of the \*PSSR.  
 (b) If there is no \*PSSR coded, an inquiry message is sent as soon as the decimal data error occurs in P3.

The response by the user to the message determines what happens next. For example, if a cancel or dump option was taken, P3 would end abnormally and return to P2. P2 would set an error condition as a result of the failure of the called program. If the user responded to the message with retry or get input, P3 would attempt to continue.

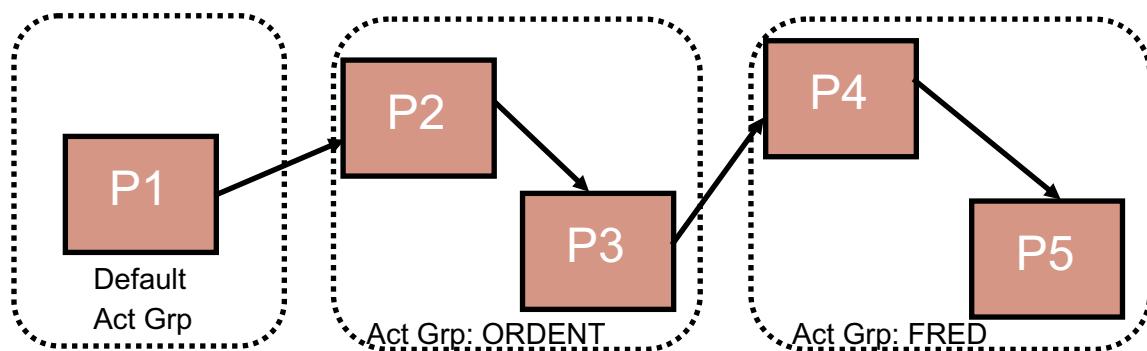
The important thing to realize is that the user of the job was notified with an inquiry message when the decimal data error occurred in P3.

# ILE control boundaries

IBM i

Percolation continues in call stack until:

- Exception is handled by a call stack entry
- or
- Control boundary is reached
- A control boundary is the first call stack entry after crossing an activation group boundary



P2 and P4 are control boundaries

© Copyright IBM Corporation 2011

Figure 7-7. ILE control boundaries

AS106.0

## Notes:

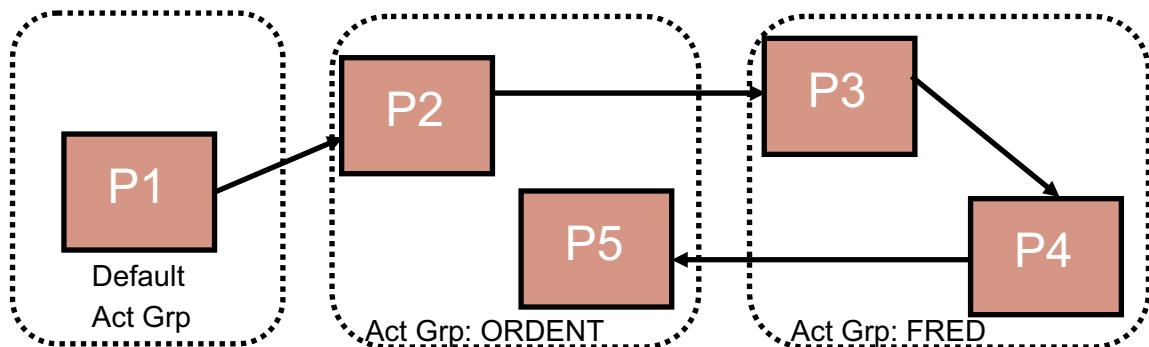
Let us take a closer look at ILE control boundaries. In the visual, the arrows represent program CALLs. Control boundaries exist in Program 2 (P2) and Program 4 (P4) because they are the first entry in the call stack after crossing the activation group boundary.

In the visual, it is not obvious how many procedures are in each program. If any of the ILE programs contain multiple procedures, each of the procedures would have a call stack entry.

For the purposes of our discussion, the number of procedures in each program is not important. However, it is important to note that for the purposes of percolation, each procedure that was called in the current invocation stack would be considered a call stack entry and could be the target of exception percolation from a later call stack entry.

# Where are the control boundaries?

IBM i



© Copyright IBM Corporation 2011

Figure 7-8. Where are the control boundaries?

AS106.0

## Notes:

How many control boundaries are there? Where are they?

# What happens at a control boundary?

IBM i

- Unhandled errors are percolated to a control boundary
  - Converted to function check
  - Sent back to original source of error
- Percolation starts again
  - RPG default handler issues inquiry message to user

© Copyright IBM Corporation 2011

Figure 7-9. What happens at a control boundary?

AS106.0

## Notes:

You have seen how percolation continues up the call stack until the error is handled or until a control boundary is reached. When an unhandled exception reaches a control boundary, it is converted to a function check and *sent back* to the procedure that caused the error in the first place. The default error handler issues an inquiry message to the user. This is similar to what happens in OPM programs now.

If you look at the message numbers, you may have noticed that the ILE messages are RNQxxxx rather than RPGxxxx. In some cases, the message text is also different for the ILE message.

# Percolation

IBM i

- Exception message generated by run-time error
  - ESCAPE, \*STATUS, \*NOTIFY
    - Function check
- Exception messages associated with call stack entries
  - Procedures are call stack entries in multiple module bound procedures
- Exception occurs!
  - Does current call stack entry handle error?
    - If not
  - Exception is percolated to previous call stack entry

© Copyright IBM Corporation 2011

Figure 7-10. Percolation

AS106.0

## Notes:

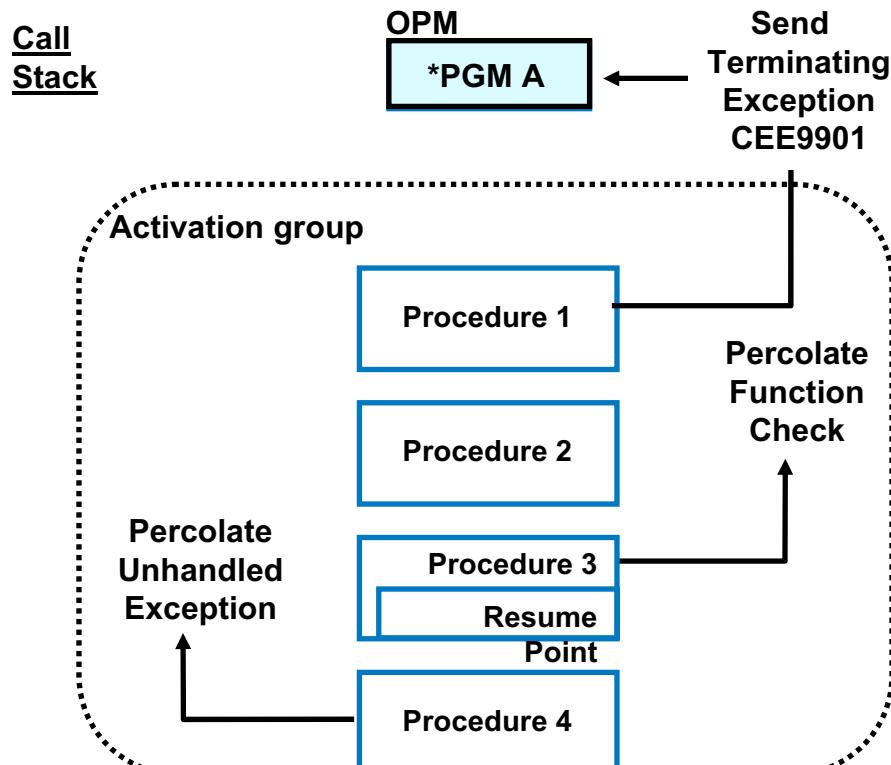
One of the main concepts of ILE exception handling is *percolation*. When a run-time error occurs, the system first checks whether your program handles the exception. If there is a *handler* that is active (for example, an E opcode extender and the %Error BIF), percolation does not occur. The exception is considered to have been *handled*.

However, if the exception is not handled by code in the procedure in which it occurs, the exception percolates up to the caller of that procedure.

Percolation occurs whether the caller is another procedure bound in the same program or a procedure in another program that called this one. During the percolation process, the system looks for some code or function that will accept responsibility for detecting and handling the error.

# Unhandled ILE exception

IBM i



© Copyright IBM Corporation 2011

Figure 7-11. Unhandled ILE exception

AS106.0

## Notes:

For ILE, an unhandled exception message is percolated to the previous call stack entry message queue.

In this example, procedure P1 is known as a *control boundary* because it is the oldest call stack entry in the activation group. Suppose that procedure P4 incurred an exception message that was unhandled. Percolation of an exception continues until either a control boundary is reached or the exception message is handled.

During percolation, the *resume point* is moved from P4 to P3 by the exception handler. The resume point is the instruction in a program where processing continues after the exception has been handled.

### Resume Point

The resume point is initially set at the first instruction following the point in the program where the error originally occurred.

If the error is percolated up the call stack, the resume point moves up the stack also. An unhandled exception is converted to a function check when it is percolated to the control boundary.

The resume point (shown in procedure P3) is used to define the call stack entry at which exception processing of the function check should continue. In ILE, the next processing step is to send the special function check exception message to this call stack entry. This is procedure P3 in this example.

You may write the logic necessary to handle the function check exception message at this point in your procedure, or, if you do choose not to handle it (maybe you did not anticipate this particular error), the function check exception message is percolated to the control boundary (worst case).

If you did handle the problem in your code, normal processing continues at the resume point and exception processing ends. If the function check message is percolated to the control boundary, ILE considers the application to have ended with an unexpected error. A generic failure exception message is defined by ILE for all languages.

This message, an *ILE function check*, is CEE9901. It is sent to the caller of the control boundary procedure.

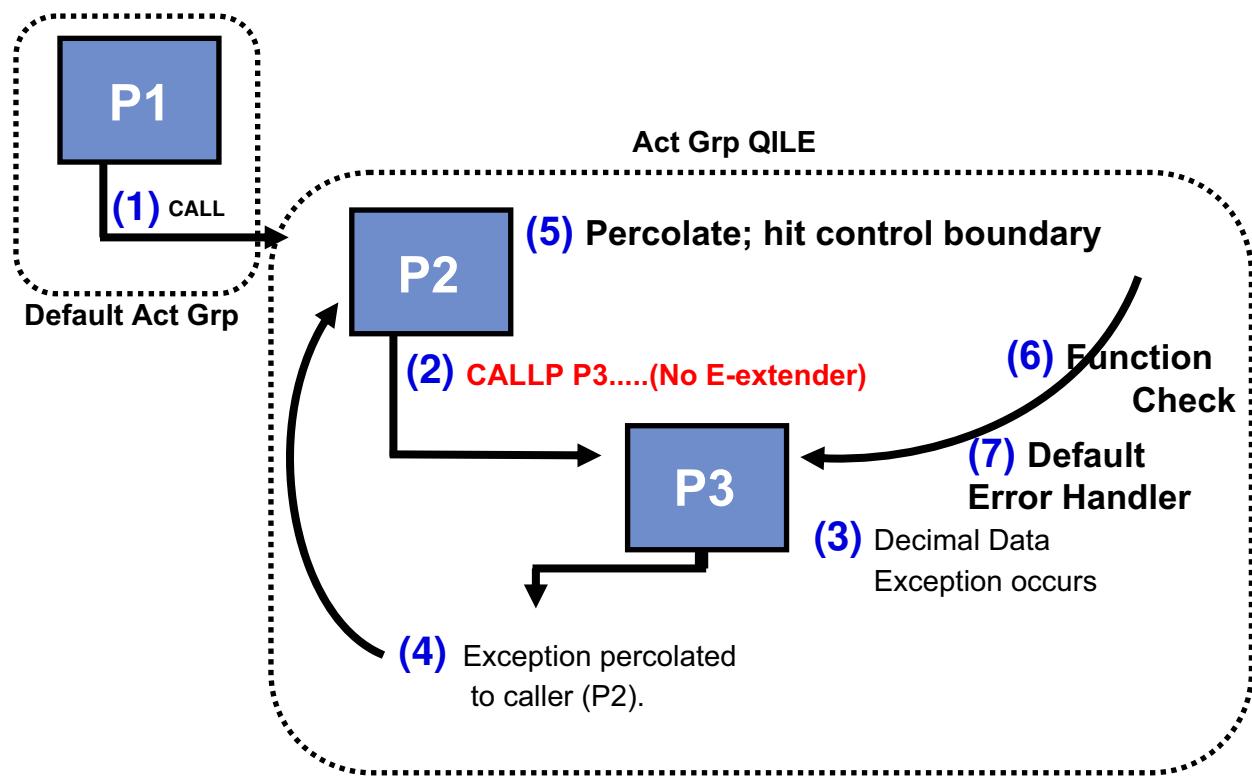
As an aside, the equivalent OPM function check message is an escape message with an ID of CPF9999.

The default action for unhandled exception messages defined in ILE allows you to recover from error conditions that occur within a mixed-language application. For unexpected errors, ILE enforces a consistent failure message for all languages. This improves the ability to integrate applications from different sources.

In either case (OPM or ILE), the user sees an inquiry message. The only differences in the previous example are when the message is issued and the prefix of the message itself (RNQxxxx instead of RPGxxxx).

# ILE exception handling example

IBM i



© Copyright IBM Corporation 2011

Figure 7-12. ILE exception handling example

AS106.0

## Notes:

Here we have re-created the earlier OPM example as an ILE application in order to illustrate how ILE error handling operates.

In the OPM visual, there was an error indicator (or E-extender) coded on the CALL in P2. We made a change to the application. We have removed the error indicator to show you one perspective of ILE error handling:

1. Program P1, running in the default activation group, calls program P2.
2. Program P2, running in QILE, calls (CALLP) program P3. Assume program P2 has no E-extender coded on the CALLP statement and that there is no \*PSSR subroutine coded in P2. In other words, we deliberately do not handle a potential error on the CALLP.
3. Program P3 encounters a run-time decimal data error. No error indicator is coded on the statement where the decimal data error occurs. No \*PSSR is coded in P3. *P3 does not handle the error.*

4. Since P3 did not handle the error, the system percolates the exception message up the call stack to P2. P2 has no \*PSSR and no error handling active on the current statement (the CALLP).
5. P2 tries to percolate the exception but encounters a control boundary.
6. Therefore, the unhandled exception is converted to a function check and is sent back to the originating call stack entry, P3.
7. The RPG IV *default error handler* issues an inquiry message for the function check to the user running the job.

## OPM versus ILE error handling

IBM i

Differences in how UNHANDLED exceptions behave:

- In OPM, P3 would issue inquiry message immediately
- In ILE:
  - Exception is percolated up the call stack
  - Control boundary encountered
  - Exception converted to function check
  - Default error handler issues inquiry message

© Copyright IBM Corporation 2011

Figure 7-13. OPM versus ILE error handling

AS106.0

### Notes:

In summary, our application example will produce the same end result. The user sees an inquiry message in either case (OPM or ILE). The only differences in our previous examples are the timing and the message itself (RNQxxxx instead of RPGxxxx).

# ILE exception handling

IBM i

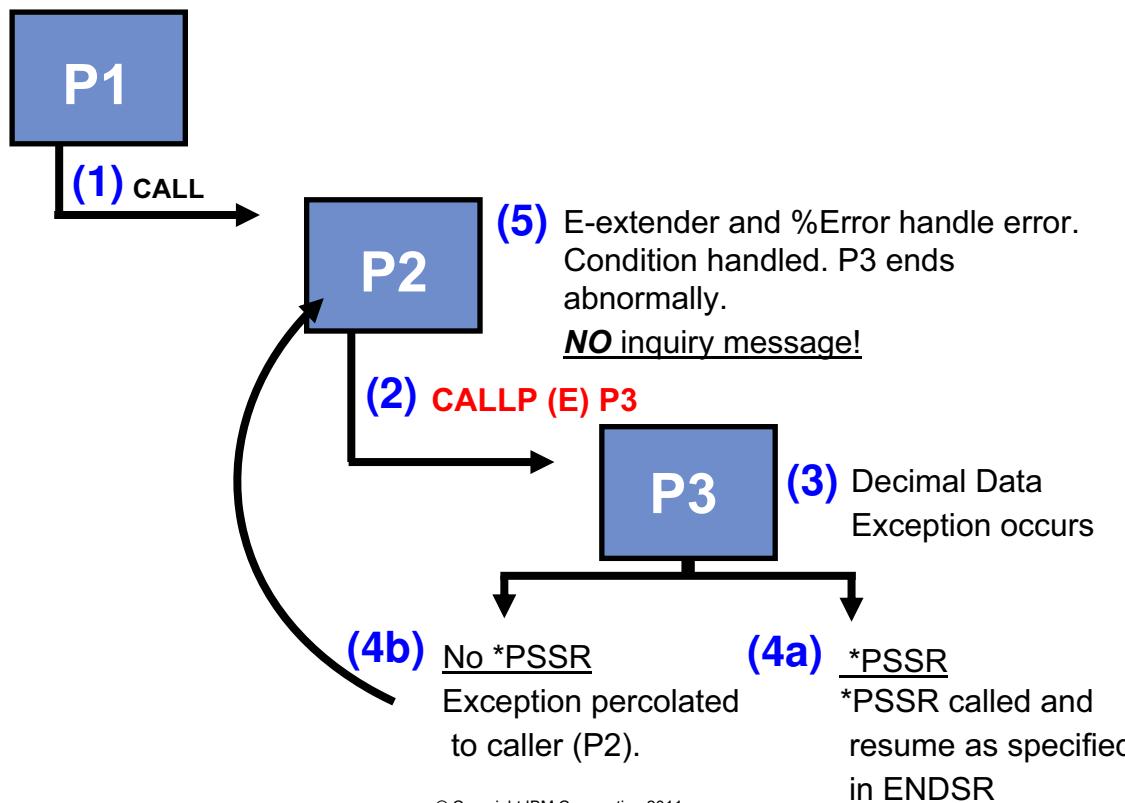


Figure 7-14. ILE exception handling

AS106.0

## Notes:

We use the same application as we did in the previous ILE example, except that we add the error indicator to the CALL in module P2. Notice how the application behaves in this situation, and compare that to the earlier OPM application:

1. Step 1 and
2. Step 2 take place as before.
3. As in the previous ILE example, when the decimal data error occurs, the system looks for a handler.
4. (a) If a handler is coded, the exception is handled and the system stops exception processing.  
(b) However, if there is no handler, such as a \*PSSR subroutine, the exception is percolated to P2.
5. This is the point where the significant difference in behavior occurs. When the exception is percolated to P2, the system finds a handler, the E-extender coded on the CALL to

P3. As far as the system is concerned, the error is handled. P3 does end abnormally, but, the E-extender and the %Error BIF in P2 tell the system that the error is handled.

Therefore, *no* inquiry message is sent to the user. The job continues processing at the statement after the CALLP in P2. If that code checks %Error and reacts in a valid manner to the termination of P3, all is well.

The behavior in this example is quite different from the behavior we discussed in the OPM version in the same situation.

# ILE error handling

IBM i

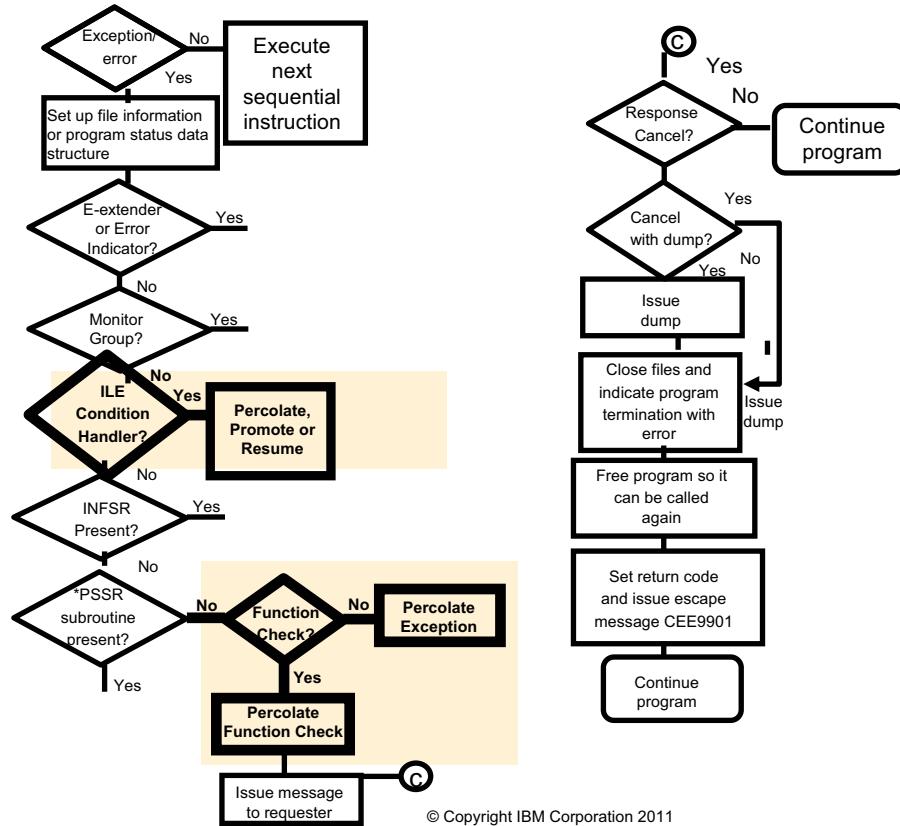


Figure 7-15. ILE error handling

AS106.0

## Notes:

Notice that the ILE condition handler is given control (if it exists for this error) in the same manner as an error subroutine. Control is not passed to the i default error handler until a function check is percolated outside the control boundary. The hierarchy of error handlers changes very little in ILE. They are ranked in order of priority, as follows. The ILE handler is highlighted:

1. Error Indicator or E opcode extender and %Error BIF handler
2. Monitor group
- 3. ILE condition handler**
4. Error subroutine (InfSR for I/O or \*PSSR for program errors)
5. RPG IV default error handler

Each of the above handlers that exists is given the opportunity to handle the error. If the handler does not exist or, if the handler does not mark the error *handled*, the error is sent to the next handler in the hierarchy above. If a handler does not exist, the process continues until it is necessary to percolate the error.

# Conversion considerations

IBM i

- Test for possible changes in application behavior
  - ILE error messages are different!
  - Why use condition handlers?
    - Flexible addition to exception handling process
    - Beyond E-extender and subroutine
    - Handles errors in opcodes not supported by E-extender (Eval)
    - Handler is a program that is external to application code can be generic (language independent)

© Copyright IBM Corporation 2011

Figure 7-16. Conversion considerations

AS106.0

## Notes:

How the differences in error handling behavior affect your code depends on the environment in which your code has been running.

# Using ILE error handlers

IBM i

- ILE condition handler
  - User-written code
  - Handles, percolates or promotes exceptions
- ILE cancel handler
  - User-written code
  - Gets control at abnormal termination
- Must be registered at run time using bindable APIs
  - CEEHDLR for condition handler
  - CEERTX for cancel handler

© Copyright IBM Corporation 2011

Figure 7-17. Using ILE error handlers

AS106.0

## Notes:

**ILE condition handlers:** User-written code to handle, percolate or promote exceptions. Percolation and promotion are basically the same process. Promotion is percolation that is managed by the application. Percolation is handled by the system.

**ILE cancel handlers:** User-written code that gets control when call stack entries are terminated abnormally, for example, when system request option 2 is taken or when C for cancel is given as a response to an inquiry message.

Both types of handlers must be *registered* at run time using bindable APIs.

- CEEHDLR API registers a procedure for a condition handler
- CEERTX registers a procedure for a cancel handler

Registration specifies the name of the procedures to get control when exception conditions or cancellation occurs (abnormal termination).

When an error is promoted, your error handler includes code that changes the message to a different message.

We focus our attention in this unit on writing a basic condition handler and percolation of messages.

# ILE condition handler

IBM i

- Handles specific or general messages
  - Procedure modifies message to handled state
  - Other options:
    - Percolate exception
    - Promote exception
- Minimum of three parameters passed to handler
  - ILE Token structure: Message ID / severity
  - Program information parameter
  - Action to take (output) such as resume or percolate

© Copyright IBM Corporation 2011

Figure 7-18. ILE condition handler

AS106.0

## Notes:

ILE condition handlers:

- Are procedures that can be coded in any ILE language.
- Are bound as modules into the program containing the procedures for which they are registered.
- Can be registered for as many procedures as desired.
- Can have more than one ILE condition handler registered for a procedure.

ILE condition handlers are registered as exception handlers at run time. ILE condition handlers are registered for a particular call stack entry. To register an ILE condition handler, use the Register a User-Written Condition Handler, (CEEHDLR) bindable API. This API allows you to identify a procedure at run time that should be given control when an exception occurs.

When it is no longer needed, an ILE condition handler can be unregistered by calling the Unregister a User-Written Condition Handler, (CEEHDLU) bindable API.

## ILE cancel handlers

IBM i

- Get control when call stack entries terminate abnormally
- Registered and unregistered like condition handlers

© Copyright IBM Corporation 2011

Figure 7-19. ILE cancel handlers

AS106.0

### Notes:

Cancel handlers allow you to get control for clean-up and recovery actions when call stack entries are terminated by an abnormal return. For example, you might want one to get control when a procedure ends via a system request 2 or because an inquiry message was answered with C (Cancel).

The Register Call Stack Entry Termination User Exit Procedure (CEERTX) and the Call Stack Entry Termination User Exit Procedure (CEEUTX) ILE bindable APIs provide a way of dynamically registering a user-defined routine to be run when the call stack entry for which it is registered is cancelled.

Once registered, the cancel handler remains in effect until the call stack entry is removed or until CEEUTX is called to disable it.



## 8.2. ILE condition handlers

# An overview of using ILE condition handlers

IBM i

1. Registered by your procedure at runtime by calling CEEHDLR
2. More than one handler can be registered for a procedure
3. Handler enables you to resume execution of procedure
4. Handler can promote or percolate error
5. Unregistered (usually by your procedure) at run-time by calling CEEHDLU

© Copyright IBM Corporation 2011

Figure 7-20. An overview of using ILE condition handlers

AS106.0

## Notes:

Remember where ILE condition handlers rank in the error handling hierarchy. For review:

1. Error Indicator or E opcode extender plus %Error BIF handler
2. Monitor group
3. ILE condition handler
4. Error subroutine (Infsr for I/O or \*PSSR for program errors)
5. RPG IV default error handler

When you write an error handler the system passes parameters that assist you in determining what the error was and how to handle it. Your handler can be designed for a specific application or can be generic, depending on the type of error. You may access more than one condition handler, but for each one you use, you must register it in the procedure that will call the handler if an error occurs.

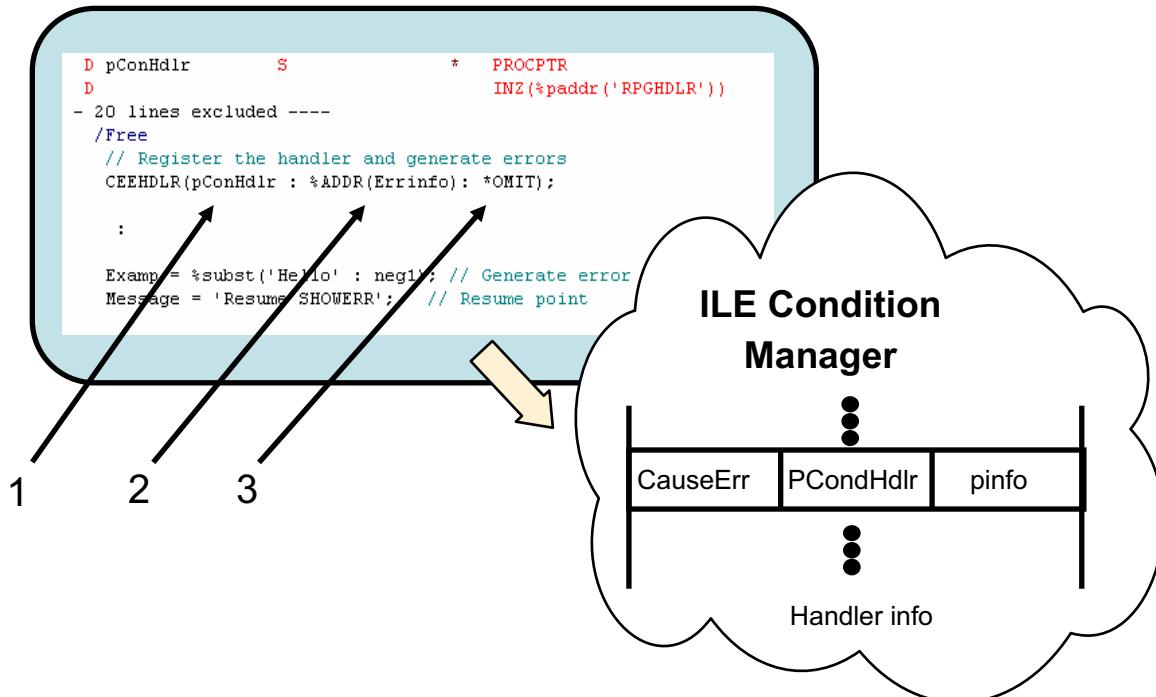
You do not *call* the handler because you have access to it once you have registered it via CEEHDLR.

You will see how the interface works soon.

# ILE condition handling: Registering the handler

IBM i

CauseErr



© Copyright IBM Corporation 2011

Figure 7-21. ILE condition handling: Registering the handler

AS106.0

## Notes:

In order to use an ILE condition handler, you must register it using the CEEHDLR bindable API. You will recall from the API unit that understanding how to format and use the parameters was the most critical part of utilizing the API.

This is also very true for the CEEHDLR API. Since CEEHDLR is the interface between your procedure and your condition handler, CEEHDLR needs to know several things in order to successfully register your handler:

1. The address of the handler itself. CEEHDLR needs to know how to find your error handler. The address of the handler must be provided using a *procedure pointer*. Procedure pointers are used to point to procedures or functions. A procedure pointer points to an entry point that is bound into the program.
2. Information about the error that was received. We explain this parameter further.
3. Optional parameter for feedback that we are not using.

Assume that we have an RPG IV program, (CauseErr), which may have some existing error handling in it. You might use RPG E-extenders, monitor groups, \*PSSR, and so on.

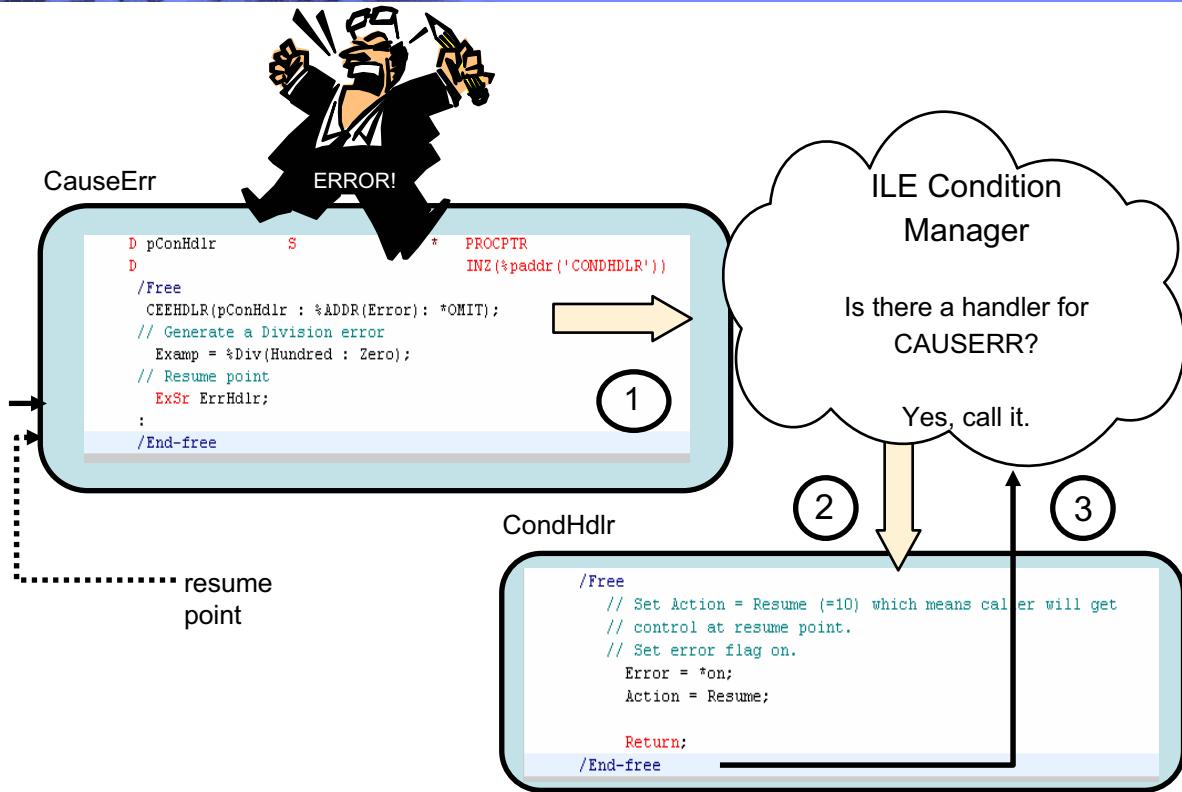
When you add an ILE condition handler to enhance your error handling, there are some steps that you must follow. This visual and the ones that follow show the components involved when you implement a condition handler. Once you have written your condition handler, your errors are handled in a manner similar to the following process. These steps apply for any error that results in a call to your condition handler:

A statement (A) to register the handler is added to the CauseErr program. A special API, CEEHDLR, is called. This API is made available to you by the system. There is no explicit bind by reference required when you create your \*PGM object. There is an implicit bind by reference to CEEHDLR completed on **CRTSRVPGM** and **CRTPGM** because the system QLEAWI \*SRVPGM is automatically included by the binder.

When CEEHDLR is called, you pass it a procedure pointer to the address (1) of the handler and a pointer to an error information data structure (2). CEEHDLR then knows that the handler that you have written will handle errors encountered by your RPG program, CauseErr. CEEHDLR is passed a pointer to the procedure CondHdlr and notes that CondHdlr handles errors in CauseErr.

# ILE condition handling: Handling an error

IBM i



© Copyright IBM Corporation 2011

Figure 7-22. ILE condition handling: Handling an error

AS106.0

## Notes:

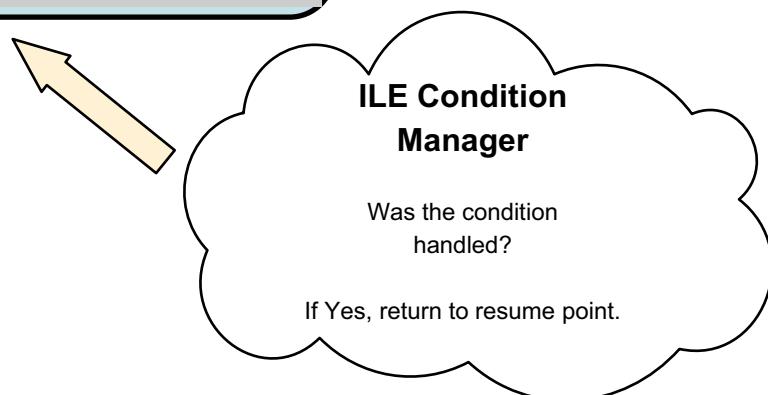
- When an error occurs in `CauseErr` that is not handled by an RPG monitor group nor by an E opcode extender, the ILE condition handler checks to see if there is a condition handler registered for `CauseErr`.
- The ILE Condition Manager sees that a condition handler is registered for `CauseErr`. `CEEHDLR` calls the handler, and, working with the ILE Condition Manager, passes error message information to the RPG condition handler.
- Once the handler has marked the error *handled*, it passes control back to the ILE Condition Manager.

# ILE condition handling: Resume point

IBM i

CauseErr

```
D pConHdlr      S          * PROCPTR
D                                     INZ(%paddr('CONDHDLR'))
/Free
CEEHDLR(pConHdlr : %ADDR(Error): *OMIT);
// Generate a Division error
Examp = %Div(Hundred : Zero);
// Resume point
ExSr ErrHdlr;
:
/End-free
```



© Copyright IBM Corporation 2011

Figure 7-23. ILE condition handling: Resume point

AS106.0

## Notes:

CauseErr resumes execution at the resume point, the next executable “line” of code.

Not shown in the visual is the prototype for CEEHdlr. For your reference, it is included below:

D CEEHDLR	PR
D pConHdlr	* PROCPTR
D CommArea	* CONST
D Feedback	12A OPTIONS (*OMIT)

**Important Note:** Resume execution means that your results may not be predictable. The resume point is the next machine instruction (MI) statement and is not the next HLL statement. The best you can do is to check the error flag. It would be dangerous to continue running program.

# Registration of condition handler

IBM i

```

D pConHdlr      S          * PROCPTR
D                                     INZ(%paddr('CONDHDLR'))
/Free
  CEEHDLR(pConHdlr : %ADDR(Error) : * OMIT);
// Generate a Division error
  Examp = %Div(Hundred : Zero);
// Resume point
  ExSr ErrHdlr;
:
/End-free

```

© Copyright IBM Corporation 2011

Figure 7-24. Registration of condition handler

AS106.0

## Notes:

The most important consideration when registering a handler with CEEHDLR is its parameters.

The visual shows the parameter definitions for the CEEHDLR API. The first parameter is the procedure pointer to the procedure which will handle the exception. The second parameter is a pointer to what is called a *communications area*, which is passed to the exception-handling procedure. These parameters must be passed in order to register and unregister a condition handler:

### 1. Register:

- The first parameter is a pointer to the handler procedure. The ILE Condition Manager needs the address of the Handler procedure. In this case, we point to the handler procedure CondHdlr.

You may notice something new in this visual. The %paddr BIF, in combination with the PROCPTR keyword, is used to point to the address of the CondHdlr condition

handler. It works for procedures in the same way as you saw it work for data items in the topic that discussed based variables and pointers.

- b. The second parameter passed to CEEHDLR is a pointer to the communications area, which means anything you want.

The second parameter is usually used to include a way for the handler to indicate that it was invoked. It also might contain more information, such as a pointer to the program status data structure of the procedure that caused the error. Remember that a condition handler can be registered for one or many procedures.

In this example, the second parameter is pointing to the error indicator in the CAUSEERR procedure. We supply this value by pointing to the error indicator in the CauseErr procedure.

- c. The third parameter is optional and is not used in this class. You could use it to signal that a handler is successfully registered.

## 2. **Unregister:** Optional, not shown in the visual

We could have added code to unregister the handler by calling CEEHDLU. However, any registered user-written condition handlers not unregistered by CEEHDLU are unregistered automatically by the system upon removal of the associated call stack entry from the call stack.

- a. The first parameter is a procedure pointer to the handler procedure. In this case, we would point to the handler procedure CondHdlr (as we did with CEEHDLR).
- b. The second parameter is optional. You could use it to signal that a handler is successfully unregistered. If you do not enter a parameter, you would enter **\*OMIT**.

The procedure CauseErr is now ready to work with the CondHdlr condition handler.

Note that CauseErr does not call CondHdlr. The call to CondHdlr is managed by CEEHDLR, the ILE Condition Manager, when it intercepts an error from CauseErr.

# Procedure interface: Condition handler

IBM i

```

FIG625.RPGLE X
Line 1      Column 1      Replace
.....1.....+.....2.....+.....3.....+.....4.....+.....5.....+.....6.....+.....7.....+.....8.....+
000100      // Prototype
000200      D CondHdlr      PR
000300      D Parm1
000400      D Parm2          * Like(CondToken)
000500      D Parm3          10I 0
000600      D Parm4          Like(CondToken)
000700
000800      // Procedure Interface
000900      D CondHdlr      PI
001000 >>1  D InToken          Like(CondToken)
001100 >>2  D pErrInfo        *
001200 >>3  D Action           10I 0
001300 >>4  D OutToken         Like(CondToken)
001400      // Pointer to allow breakup of token in CondToken DS
001500      D InTokPoint     S          *
001600
001700      D CondToken       DS          BASED(InTokPoint)
001800      D CondSev          5I 0
001900      D CondMsgNo        2A
002000      D
002100      D CondPrefix       1A
002200      D

```

© Copyright IBM Corporation 2011

Figure 7-25. Procedure interface: Condition handler

AS106.0

## Notes:

When an error is detected in the procedure CauseErr, there are four parameters that are passed via the ILE Condition Manager to the condition handler, CondHdlr, that is registered to handle errors in CauseErr.

1. The first parameter is passed as an input only parameter. This parameter is called the **Condition Token**.

This 12-byte parameter contains subfields that allow you to analyze the error message and determine what action you should take. The DS defining the Condition Token is shown, but we analyze the Condition Token in the next visual.

2. The second parameter is a pointer to the error information area in the procedure that registered the handler with CEEHDLR. In our example, we point to the address of the error indicator in the procedure where the error occurred, CauseErr. This parameter is an input-only parameter.

3. The third parameter is set in your handler. It contains a code representing the action that you have taken to handle the error in your condition handler, CondHdlr. We also analyze this parameter in a subsequent visual.
4. You may optionally create a new condition token. This parameter is necessary only when you *promote* the message. Promotion means that not only are you handling the error, but you are replacing the error message with one of your own.

# Condition token

IBM i

Subfield	Defined As	
ILE Condition ID		
	Message Severity	5-digit Integer
	Message Number	2-Character
Case/Severity/Control	1-Character	
Facility ID	3-Character	
I_S Information	4-Character	

```

003800      D  InTokPoint      S          *
003900
004000      DCondToken       DS          Based(InTokPoint)
004100      D  CondSev        5I 0
004200      D  CondMsgNo     2A
004300      D
004400      D  CondPrefix     1A
004500      D
-----          3A
                         4A

```

© Copyright IBM Corporation 2011

Figure 7-26. Condition token

AS106.0

## Notes:

This visual shows you the layout of the ILE condition token. Below, we explain the contents of each field and any subfields. Not all fields are used in every procedure. There are some fields that are designed to allow flexibility in the design of your error-handling architecture.

- **Condition\_ID:** A 4-byte identifier that, with the Facility\_ID, describes the error condition that the token communicates. The Condition\_ID contains:
  - Message Severity ( \* ): A 5-digit integer that indicates the severity of the error passed in the condition token. This subfield and the **Severity** field contain the same information. The following visual describes message severities.
  - Message Number ( \* ): A 2-byte binary number (can be defined as character) that identifies the message associated with the condition. The combination of *Facility\_ID* and *Msg\_No* uniquely describe a condition.
- **Case/Severity/Control:**
  - Case: A 2-bit field that defines the format of the Condition\_ID portion of the token. ILE conditions are always case 1.

- Severity: A 3-bit binary integer that indicates the severity of the condition. The **MsgSev** field contains the same information. The next visual will describe message severity.
- Control: A 3-bit field containing flags that describe or control various aspects of condition handling. The third bit specifies whether the Facility\_ID was assigned by IBM.
- **Facility\_ID ( \* )**: Sometimes referred to as the *message type*, the Facility\_ID is a 3-character alphanumeric string that identifies the source of the message. The Facility\_ID indicates whether the message was generated by the system or by an HLL run time procedure. For RPG IV errors, the Facility\_ID contains the RPG IV message prefix, for example, RNX.
- **I\_S Information**: A 4-byte field that identifies the specific information associated with a given instance of the condition. This field contains the reference key to the instance of the message associated with the condition token and is often called the **message handle**. It can be used to retrieve the message from the message stack.

#### \* RPG IV parameters

For most RPG IV applications, the subfields that you use most often are the **Message Severity**, **Message number**, and **Facility\_ID**.

# Error message severity

IBM i

From IBM i Message Severity	TO ILE Condition Severity	To OS/400 Message Severity
*ESCAPE messages		
0-29	2	20
30-39	3	30
40-49	4	40
*STATUS & *NOTIFY messages		
0	0	0
1-99	1	10

© Copyright IBM Corporation 2011

Figure 7-27. Error message severity

AS106.0

## Notes:

We are all familiar with error severity in IBM i. This table shows you the correspondence between IBM i severity and ILE condition severity.

A message is associated with every condition that is raised in ILE. The token contains a unique ID that ILE uses to write a message associated with the condition to the message file.

The format of every run-time message is FFFxxxx, where:

**FFF:** The facility ID, a 3-character ID that is used by all messages generated by ILE and ILE languages. When checking the facility ID in your handler, the ILE message prefix in RPG is RNX. Your handler would check for messages prefixed RNX rather than MCH.

**xxxx:** The error message number. This is a hexadecimal number that is the error message associated with the condition. The error message number corresponds to the four-digit number associated with the system message; for example, **1202** is a division by zero error.

If you did not have a registered handler for a procedure, and that procedure experienced a divide by zero error, the joblog would contain the MCH1202 error message. If you registered a handler and wrote the code to handle this error, you would look for ILE error message RNX1202.

# Actions passed to ILE Condition Manager

IBM i

- Resume (10)
  - Mark error handled
  - Set action to resume
- Percolate (20, 21)
  - Do not mark error as handled
  - Set action to percolate
- Promote (30, 31, 32)
  - Mark error handled
  - Set action to promote
  - Create new condition token calling CEENCOD

© Copyright IBM Corporation 2011

Figure 7-28. Actions passed to ILE Condition Manager

AS106.0

## Notes:

If you determine that processing can continue, you can specify that execution is to continue at the resume point in the procedure that experienced the error. Before you can resume processing, you must change the exception message to indicate that it has been handled.

For an ILE RPG IV condition handler, you continue processing by setting an action code and returning to the system.

Valid action codes are:

**Resume:** This result code can be used for all exception types.

**10:** Resume at the resume point and handle the condition, as follows:

- Function Check (severity 4): The message appears in the job log.
- \*ESCAPE (severity 2-4): The message appears in the job log.
- \*STATUS (severity 1): The message does not appear in the job log.
- \*NOTIFY: The default reply is sent and the message appears in the job log.

**Percolate:** These result codes can be used for all exception types.

**20:** Percolate to the next condition handler.

**21:** Percolate to the next call stack entry.

**Promote:** Only \*ESCAPE and \*STATUS messages may be promoted.

**30:** Promote to the next condition handler.

**31:** Promote to the next call stack entry.

**32:** Promote and restart condition handling.

When writing your RPG IV condition handlers, a good idea is to code these actions as named constants. In this way, your code is more readable.

# Condition handler: CONDHDLR (1 of 2)

IBM i

```

000001      H NoMain
000002
000100      // CondHdlr Prototype
000200      D CondHdlr          PR
000300      D Parm1             12A
000400      D Parm2             *
000500      D Parm3             10I 0
000600      D Parm4             12A
000601
000602      PCondHdlr         B           Export
000700      // CondHdlr Interface
000800      D CondHdlr         PI
000900 >>1  D InToken          Like(CondToken)
001000 >>2  D pErrInfo        *
001100 >>3  D Action           10I 0
003900
004000 >>a DCondToken        DS           Based(InTokPoint)
004100      D CondSev          5I 0
004200      D CondMsgNo        2A
004300      D                 1A
004400      D CondPrefix        3A
004500      D                 4A
004501

```

© Copyright IBM Corporation 2011

Figure 7-29. Condition handler: CONDHDLR (1 of 2)

AS106.0

## Notes:

This is the first of two visuals that shows the D-specs for the condition handler, CondHdlr. This procedure's parameters are:

1. Condition token structure (Input): Note the use of a pointer to break the condition token into its subfields.
2. Pointer to the communications area (Input): This communications area really means that we can **point to anything** we want. In our example, we simply point to an error indicator that tells us whether we did or did not handle an error.
3. A code that contains the action to be performed on the exception (Output).
4. New condition token if we decide to promote the condition (Output). Since this handler only resumes and percolates, we ignore this parameter.

In the visual, notice the prototype and the procedure interface for this procedure. Because we want to subdivide the input condition token, we define a data structure CondToken (a) that is based on a pointer InTokPoint (e). When we begin execution, we place the address

of the condition token InToken in InTokPoint (e). We want access to the error indicator (b) (that is also in CauseErr).

The Action codes Resume (c) and Percolate (d) are coded as constants so that a more meaningful name can be used in the calculations.

## Condition handler: CONDHDLR (2 of 2)

IBM i

```

004600      /Free
005000          Message = 'Starting CondHdlr';
005100          Dsply Message '*REQUESTER';

005101
005901      // Set error flag on to indicate handled.
005902 >>1      Error = *on;
005903      // Set Action = Resume (10). Procedure with error will get
005904      // control at resume point.
006000 >>2      Action = Resume;

006001
006101      Return;
006102      /End-Free
006200      P          E

```

© Copyright IBM Corporation 2011

Figure 7-30. Condition handler: CONDHDLR (2 of 2)

AS106.0

### Notes:

The logic of this procedure does the following:

1. Sets the error indicator to show that the error has been handled.
2. Sets the action code to resume.

In order that you can see control being passed to and from CondHdlr, we use the DSPLY opcode to post messages to your job queue.

This handler does not check any message codes. It simply marks each and every message as handled by setting the action code to resume. We return control to the ILE Condition Manager, which then allows CauseErr to resume execution at the resume point.

# Main procedure: CauseErr (1 of 2)

IBM i

```

000101
000200 >>2 D pConHdrl      S          * PROCPTR
000300   D                                     INZ (%paddr ('CONDHDLR'))
000301
000400 >>3 D Error       S          1N    inz(*off)
000401
000500   // Array that will be used to cause an error
000600   D Arri        S          10A   DIM(5)
000601
000700   // Message field to display status in MSGQ
000800   D Message      S          40A   INZ
000801
000900   // Work fields used to generate errors
000901   D Hundred      S          5     0 INZ(100)
001000   D Zero         S          5     0 INZ(0)
001100   D Examp        S          5     0 INZ
001101
001200   // CEEHDLR Interface
001300 >>1 D CEEHDLR     PR
001400   D pConHdrl      S          * PROCPTR
001500   D CommArea      S          * CONST
001600   D Feedback       S          12A   OPTIONS(* OMIT)
001601

```

© Copyright IBM Corporation 2011

Figure 7-31. Main procedure: CauseErr (1 of 2)

AS106.0

## Notes:

These are the data definition specifications for CauseErr. This program generates some deliberate errors that are trapped by a condition handler, CondHdrl.

The major elements of the D-specs are:

1. Prototype for the parameter interface to CEEHDLR, the registration API for the ILE Condition Manager. There are three parameters: a pointer to the address of the condition handler, a communications area that we use to pass some error information between CauseErr and the handler, and a third parameter that we do not need.
2. pConHdrl is a pointer that contains the address of the condition handler, CondHdrl. This is the first parameter that we pass to CEEHDLR. You notice the procedure pointer in this visual once again. The %paddr BIF in combination with the PROCPTR keyword is used to point to the address of the CondHdrl condition handler.
3. You recall the error information communications area. We pass its address to CEEHDLR and use it to send and receive information about the status of the error that is being handled.

## Main procedure: CauseErr (2 of 2)

IBM i

```

001605      /Free
001607          Message = 'Starting CauseErr';
001609          Dsply Message '*REQUESTER';
001610
001611      // Register the handler and generate errors
001612 >>1      CEEHDLR(pConHdlr : %ADDR(Error): * OMIT);
001613
001614          Message = 'In CauseErr - causing Division error';
001615          Dsply Message '*REQUESTER';
001616
001617      // Generate a Division error
001618 >>2      Examp = %Div(Hundred : Zero);
001619      // If the exception was handled by the handler (Action code = 10)
001620      // then execution resumes here.
001621          Message = 'Resume CauseErr';
001622          Dsply Message '*REQUESTER';
001623 >>3      ErrHdl;
001624
001647
001648      *Inlr = *on;
001649 >>5      Return;
001650
001651      //-----Subroutines-----
001652 >>4      BegSr ErrHdl;
001653      If Error;
001654          Message = 'In CauseErr Subr - error detected';
001655          Dsply Message '*REQUESTER';
001656          Error = *Off;
001657          //           Else
001658          // What if error not equal *on?
001659          EndIf;
001660          EndSr;
001661
001662      /End-Free

```

© Copyright IBM Corporation 2011

Figure 7-32. Main procedure: CauseErr (2 of 2)

AS106.0

### Notes:

The basic logic of the procedure CauseErr is:

1. Register the handler, CondHdlr, with the ILE Condition Manager by making a prototyped call to the CEEHDLR API.
  2. Generate a divide by zero error.
- When the error is encountered, the handler CondHdlr is automatically called. It handles the exception and indicates that processing should resume.
3. Processing resumes at the next instruction following the statement that caused the error.
  4. In the subroutine, check if error = \*on. If true, then CondHdlr has seen this error and handled it.
  5. Return.

# Creating PGM with condition handler

IBM i

- Create modules from procedure source:
  - CRTRPGMOD MODULE (CONDHDLR)
  - CRTRPGMOD MODULE (CAUSEERR)
- Bind modules into \*PGM (bind by copy or reference)
  - CRTPGM PGM(ILECOND) MODULE (CAUSEERR CONDHDLR)
  - Do not have to bind CEEHDLR
- Run the application
  - CALL PGM(ILECOND)

Implement generic error handling by placing error handlers in \*SRVPGM

© Copyright IBM Corporation 2011

Figure 7-33. Creating PGM with condition handler

AS106.0

## Notes:

In order to create a \*PGM object that contains both the application procedure and the condition handler, you must first create the modules.

The condition handlers can be bound by copy or stored in a \*SRVPGM and bound by reference. Notice that it is not necessary to bind the CEEHDLR API.

**Note:** We recommend that you place error handling modules such as CONDHDLR in a \*SRVPGM. *This is the real power of ILE condition handlers!* Generic error handling procedures can be built external to your application code and are called automatically. If the handler needs to be enhanced or extended, it does not affect any of the code in which it is registered.

# ILE condition handling: The process in review

IBM i

- Register the handler with the ILE Condition Manager
  - What is the address of the handler procedure?
  - What is the address of the information you want to pass?
- ILE Condition Manager can now associate your application with handler
- If error occurs and it is not handled by E-extender or %Error, ILE Condition Manager checks for registered handler
  - Handler expects error and marks it handled (resume)  
-OR-
  - Handler does not expect error and marks it percolate
- Return to ILE Condition Manager
  - Condition Manager returns control to application resume point  
-OR-
  - Condition Manager returns control (and error) to application and application percolates

© Copyright IBM Corporation 2011

Figure 7-34. ILE condition handling: The process in review

AS106.0

## Notes:

When you consider using the handler from the process perspective, it seems fairly straightforward. Relatively little code needs to be added to use a handler, and the handler itself is written using RPG IV or any other ILE language.

The most complex part of using a handler is the same thing that can cause problems when you call APIs: the parameters.

## Condition handler: Magic?

IBM i

- NO! Does not fix problems!
- Does:
  - Trap errors
  - Can assist you to isolate errors
- Problems must be resolved by you
- Use condition handlers to enhance traditional error handling

© Copyright IBM Corporation 2011

Figure 7-35. Condition handler: Magic?

AS106.0

### Notes:

Handlers were not meant to be able to perform magic and make errors go away. They were designed to give you the ability to trap (monitor) errors in RPG IV and do something about them. In this context, a handler can be an excellent tool to help diagnose the cause of a problem by isolating it to a certain type of error or to a certain statement.

In a production environment, this can be a great tool. Not only can a condition handler isolate the source of the problem, it also isolates the problem from the end user.

Remember, a handler does *not* make the error go away. It traps it for you. Also, perhaps as you become more skilled in using handlers, it may be possible to write code to intercept some errors on the fly.

Using a condition handler can enable you to get control before the default error handler gets control, and even handle the exception and continue processing.

Condition handlers should not completely replace language-specific error handling; they just give you more flexibility. A condition handler could allow you to change the message that is issued. For example, if the exception is MCH1202 (decimal data error), you could

change it to your own message such as *ABC0101: An unexpected error occurred. Please contact your ABC Software representative.*

The joblog would show this new message instead of the MCH1202 message.

# Machine exercise: Enhancing the condition handler

IBM i



© Copyright IBM Corporation 2011

Figure 7-36. Machine exercise: Enhancing the condition handler

AS106.0

## Notes:

Perform the machine exercise *enhancing the condition handler*.

## Checkpoint

IBM i

1. True or False: A control boundary is the first call stack entry after crossing an activation group (AG) boundary.
  
2. During ILE program execution, unhandled errors or exceptions:
  - a. Go to the system default error handler.
  - b. Cause the program to immediately fail.
  - c. Are percolated up the call stack until it handled or it reaches a control boundary.
  - d. Behave just like errors in OPM programs.
  
3. True or False: An ILE condition handler is a system provided API that can handle, percolate or promote exceptions in an ILE program.

© Copyright IBM Corporation 2011

Figure 7-37. Checkpoint

AS106.0

### Notes:

## Unit summary

IBM i

Having completed this unit, you should be able to:

- Describe the hierarchy of error handling in ILE
- Define and differentiate promote and percolate
- Describe how to use an ILE condition handler
- Code an RPG IV ILE condition handler

© Copyright IBM Corporation 2011

Figure 7-38. Unit summary

AS106.0

### Notes:

# Unit 9. Other RPG IV compiler features

## What this unit is about

This unit highlights some of the enhancements made to the RPG IV compiler that have made the language more versatile.

## What you should be able to do

After completing this unit, you should be able to:

- Use (with documentation) the enhancements described in the lecture material.

## Unit objectives

IBM i

After completing this unit, you should be able to:

- Use (with documentation) the enhancements described in the lecture material.

© Copyright IBM Corporation 2011

Figure 8-1. Unit objectives

AS106.0

### Notes:

After completing this unit, you should be able to use (with documentation) enhancements to the RPG compiler.

## V6R1 enhancements

IBM i

- MAIN keyword on the Control (H) specification
- Files defined in subprocedures
- Files defined like other files
- Increased size limits
  - Data structures (DIM and OCCURS)
  - Character variables
  - UCS-2 and graphic variables
- Eliminate unused variables from the compiled object
- EXFMT can use a data structure
- Alphanumeric, UCS-2 and graphic variables compared without explicit conversion



© Copyright IBM Corporation 2011

Figure 8-2. V6R1 enhancements

AS106.0

### Notes:

The following list describes the enhancements made to ILE RPG in V6R1:

- **MAIN(main\_procedure\_name)** – This Control specification keyword specifies the name of a subprocedure that will act as the program-entry procedure when the program is called. This technique allows you to have programs consisting completely of modules without the RPG logic cycle, i.e. additional modules could be coded with subprocedures using the NOMAIN keyword. Below is an example:

**H MAIN(DisplayCurTime)**

\* Prototype for the program

**D DisplayCurTime PR EXTPGM('DSPCURTIME')**

\* -----

\* Procedure name: DisplayCurTime

\* -----

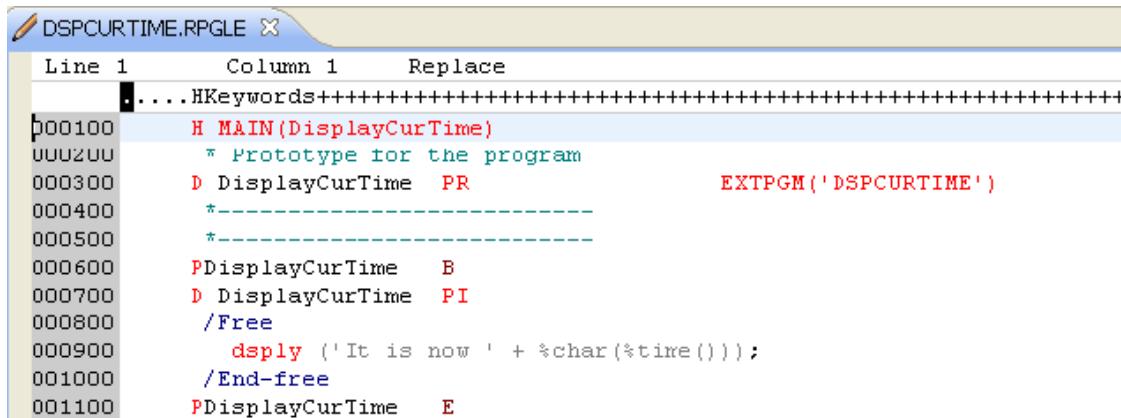
**P DisplayCurTime B**

**D DisplayCurTime PI****/FREE****dsply ('It is now ' + %char(%time()));****/END-FREE****P DisplayCurTime E**

- **Files defined in subprocedures** – Files can be defined locally in subprocedures and I/O to these locally defined files is done with data structures, since Input and Output specifications are not allowed in subprocedures.
- **Files defined like other files** – Using the LIKEFILE keyword, a file can be defined to use the same settings as another File specification, which is important when passing a file as a parameter. If the file is externally-described, the QUALIFIED keyword is implied. I/O to the new file can only be done through data structures.
- **Increased size limits for variables** – Data structures can have a size up to 16,773,104. Character variable definitions can have a length up to 16,773,104. (The limit is 4 less for variable length character definitions.) UCS-2 definitions can have a length up to 8,386,552 UCS-2 characters. (The limit is 2 less for variable length UCS-2 definitions.) Graphic definitions can have a length up to 8,386,552 DBCS characters. (The limit is 2 less for variable length graphic definitions.) The VARYING keyword (with a parameter of either 2 or 4 indicating the number of bytes used) to make the variable varying in length and the system will hold the length in an appropriate (2 or 4 byte) prefix.
- **Eliminate unused variables with \*NOUNREF** – There are new values added to the OPTION keyword for the CRTBNDRPG and CRTRPGMOD commands, and for the OPTION keyword on the Control specification - \*UNREF and \*NOUNREF. The default is \*UNREF. \*NOUNREF indicates that unreferenced variables should not be generated into the RPG module. This can reduce program size, and if imported variables are not referenced, it can reduce the time taken to bind a module to a program or service program.
- **EXFMT with a data structure in the result field** – The EXFMT operation was enhanced to allow a data structure to be specified in the result field. The data structure must be defined with usage type \*ALL, either as an externally-described data structure for the record format (EXTNAME(file:fmt:\*ALL), or using LIKEREC of the record format (LIKEREC(fmt:\*ALL).
- **Implicit conversion between character, UCS-2 and graphic values** – Implicit conversion is now supported for the following: assignment using EVAL and EVALR, comparison operations in expressions, and the fixed-form comparison operations IFxx, DOUxx, DOWxx, WHxx, CASxx, CABxx and COMP. UCS-2 variables can be initialized with character or graphic literals without using the %UCS2 built-in function.

# Main subprocedure

IBM i



```

Line 1      Column 1      Replace
. ....HKeywords+++++++++++++++++++++
b00100    H MAIN(DisplayCurTime)
UUU2UU    * Prototype for the program
000300    D DisplayCurTime  PR          EXTPGM('DSPCURTIME')
000400    -----
000500    -----
000600    PDisplayCurTime   B
000700    D DisplayCurTime  PI
000800    /Free
000900    dsplay ('It is now ' + %char(%time()));
001000    /End-free
001100    PDisplayCurTime   E

```

© Copyright IBM Corporation 2011

Figure 8-3. Main subprocedure

AS106.0

## Notes:

The program is named DSPCURTIME, and its one module has a linear-main procedure called DisplayCurTime. The Control specification MAIN keyword signifies that this is a linear-main module, and identifies the name of the procedure within the module that is the special subprocedure which serves as the linear-main procedure, which will act as the program-entry procedure. The prototype for the linear-main procedure must have the EXTPGM keyword with the name of the actual program. This program/module was compiled without incorporating the RPG logic cycle and because it uses a subprocedure, could not use the default activation group.

# Files in subprocedures

```

*ITEMINQMN.RPGLE X
Line 1      Column 1      Replace 4 changes
.....HKeywords+++++-----+
000001 H MAIN(GetItemInf)
000002 DGetItemInf      PR          ExtPgm('ITEMINQMN')
000003 D               5
000004 PGetItemInf      B
000005 1   FItem_PF    IF   E      K Disk
000006
000007 2   D GetItemInf      PI
000008 D  ItmNbrPass      5
000010
000011 3   D ItemData      DS      LikeRec(Item_Fmt)
000012 D ItemMsg         S      20A Varying
000013 D ItmNbrCh        S      5 0
000014 /FREE
000018
000019     ItmNbrCh = %dec(ItmNbrPass:5:0);
000024 4   Chain ItmNbrCh Item_PF ItemData;
000026
000027     If %found(Item_PF); // Item Number valid?
000028       ItemMsg = %trim(%char(ItemData.itmnbr) + ' found.');
000030     Else;
000031       ItemMsg = %trim(%char(ItmNbrCh) + ' not found!');
000032   Endif;
000033
000034   // Display message with error or not
000035   dsplay ('Item number ' + ItemMsg);
000037
000038   *InLR = *on;
000039
000040   /End-Free
000042 PGetItemInf      E

```

© Copyright IBM Corporation 2011

Figure 8-4. Files in subprocedures

AS106.0

## Notes:

The basic logic of the procedure **GetItemInf** is:

1. The Item\_PF is defined to the subprocedure. In keeping with normal RPG specification sequence, these F-specs must come before the subprocedure's Procedure Interface (PI) definition (at 2), since in RPG, F comes before D! This causes the P-spec that marks the start of the subprocedure to be separated (by the F-spec) from the PI.
2. The PI for the subprocedure is defined. Note that we have defined the ItmNbr variable that will be passed to the subprocedure.
3. In subprocedures there is no support for I or O specs, and therefore none are generated for the file defined in the subprocedure. As a result, all I/O must be done using the result-field data structure. We therefore define a data structure to match the record layout that we require using the LIKEREC keyword.
4. The CHAIN operation uses ItmNbrCh to find the item in the Item\_PF file and the ItemData data structure is specified as the result field to receive the record. Note that the ItemData data structure is implicitly qualified, since it was defined with the LIKEREC

keyword. So the full name of the itmnbr field in the data structure is ItemData.itmnbr, and this is the name used as part of the record found message.

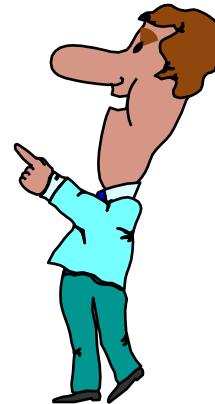
Subprocedures allocate the storage for their variables when they are called, and release it when they return or end. The same thing happens with files. They are opened when the subprocedure is called, and closed when it exits. Opening and closing files can be expensive in terms of system overhead.

One answer might be to specify the keyword STATIC on the file's F-spec. When this is done, the file will be opened the first time the subprocedure is called and remain open until explicitly closed or the activation group ends. The STATIC keyword used with the USROPN keyword would allow you to have full control of the file. Do not forget to add the STATIC keyword to the data structure if you want to retain record data between subprocedure invocations.

# V7R1 enhancements

IBM i

- %SCANRPL and %LEN BIFs
- RPG Open Access
- Optional prototypes
- Better sorting and searching arrays
  - Data structure arrays sorted/searched by key subfields
  - SORTA ascending or descending
- Alias names used for externally described DS
- Teraspace storage model
- Encrypted listing debug view



© Copyright IBM Corporation 2011

Figure 8-5. V7R1 enhancements

AS106.0

## Notes:

The following list describes the enhancements made to ILE RPG in V7R1:

- **%SCANRPL** – The %SCANRPL built-in function can find all of the occurrences of the search value in a string and replace them with another value. You can specify a start and length for the scanning phase of the operation. The resulting string includes all of the text of given string, with the replacement occurring within the bounds of the start and length. Following is the syntax of the built-in function and an example of its use:

```
%SCANRPL(scan string : replacement : source { : scan start { : scan length } } )
// Replace NAME with 'Tom'
string1 = 'See NAME. See NAME run. Run NAME run.';
string2 = %ScanRpl('NAME' : 'Tom' : string1);
// string2 = 'See Tom. See Tom run. Run Tom run.'
```

- **%LEN** – The %LEN built-in function has been enhanced to return the maximum number of characters for a varying-length field. Below is the syntax of the built-in function and a sample of its use:

`%LEN(varying-length expression : *MAX)`

```
D char_varying    s      100a  varying
D ucs2_varying   s      5000c  varying
D graph_varying  s      7000g  varying(4)
D graph_fld10    s      10g
D char_fld10     s      10a

/free

// Calculate several length and size values
// - The maximum length, %LEN(*MAX), measured in characters
// - The current length, %LEN, measured in characters
// - The size, %SIZE, measured in bytes, including the
// 2- or 4-byte length prefix
// Each alphanumeric character has one byte
char_varying = 'abc';

// Length is 3
max_len = %len(char_varying : *MAX);
len = %len(char_varying);
size = %size(char_varying);

// max_len = 100
// len = 3
// size = 102 (100 + 2)
```

- **RPG Open Access** – The concept of Open Access is one that uses a user-provided program or procedure (called a “handler”) to handle the I/O operations (from an RPG program) rather than using the system’s data management functions. The Open Access handler receives all the information it needs for the I/O operation, and it provides all the necessary feedback for the RPG program to run. Imagine an application where all of the display files are changed to be Open Access WORKSTN files. The handler procedure could interact with a web browser. The coding of the RPG program would remain traditional while the application would be enhanced with web access.

- **Optional prototypes** – If you have a program or procedure that is not called by another RPG module (not exported from the local module), you can skip coding a prototype. The compiler can generate an internal prototype from the procedure interface (PI) or if there is no PI, the compiler can generate an internal prototype with no return value and no parameters. When you don't code a prototype, you code the EXTPROC or EXTPGM keyword on the PI instead of the PR. The prototype may be omitted for the following types of programs or procedures:
  - A program that is only intended to be used as an exit program or as the command-processing program for a command
  - A program that is only intended to be called from a different programming language
  - A procedure that is not exported from the module
  - A procedure that is exported from the module but only intended to be called from a different programming language
- **Sorting and searching arrays** – Now it is possible to search and sort a data structure array using one of the subfields as a key. You also have the ability to sort an array in ascending or descending order by using the (A) or (D) extender for SORTA. If you have used the ASCEND or DESCEND keyword when defining the array, you cannot change the order of the sort using the (A) or (D) extender. This means that if you want to be able to sort the array in both ascending and descending orders and you want to be able to search for less-than or greater-than conditions, then you must decide which is more important. To search for a greater-than or less-than condition, you must code the ASCEND or DESCEND keyword. If you need the flexibility of sorting the array in either ascending or descending sequence, then use SORTA with the (A) or (D) extenders, without coding the keywords.
- **Support for alias names in data structures** – Use the ALIAS keyword on a D-spec to indicate that you want to use the alternate names for the subfields of externally-described data structures. Use the ALIAS keyword on a File specification to indicate that you want to use the alternate names for LIKEREc data structures defined from the records of the file. You can only see the alias names in data structures. You can only code ALIAS for a file that doesn't have I- and O-specs generated, which is any local file defined within a subprocedure or a file defined with the QUALIFIED keyword.
- **Teraspace memory allocations** – RPG modules and programs can be created to use the terospace storage model or to inherit the storage model of their caller. With the terospace storage model, the system limits regarding automatic storage (16 MB) are significantly higher (4 TB) than that for the single-level storage model. There are limits for the amount of automatic storage for a single procedure and for the total automatic storage of all the procedures on the call stack. You can specify STGMDL(\*TERASPACE) on the compile command or the H-spec. You also can request that your \*CALLER program or service program inherit the storage model of

its caller by specifying STGMDL(\*INHERIT) on the CRTBNDRPG, CRTPGM or CRTSRVPGM command.

- **Encrypted listing debug view** – When a module's listing debug view is encrypted, the listing view can only be viewed during a debug session when the person doing the debugging knows the encryption key. This enables you to send debuggable programs to your customers without enabling your customers to see your source code through the listing view. Use the DBGENCKEY parameter on the CRTRPGMOD, CRTBNDRPG, or CRTSQLRPGI command.

## Unit summary

IBM i

Having completed this unit, you should be able to:

- Use (with documentation) the enhancements described in the lecture material.

© Copyright IBM Corporation 2011

Figure 8-6. Unit summary

AS106.0

### **Notes:**

# Appendix A. Checkpoint solutions

## Unit 2

### Checkpoint solutions (1 of 2)

IBM i

1. Application programming interface (API) programs
  - a. Can utilize and return information in the form of parameters
  - b. Have always been available
  - c. Offer improved performance and flexibility over familiar CL commands
  - d. All of the above

The answer is all of the above.
2. In the API parameter tables, Char(\*) represents
  - a. Character data
  - b. Graphics data
  - c. Character data of unknown length
  - d. Double-byte character set (DBCS) data

The answer is character data of unknown length.
3. The format parameter of retrieve and list APIs
  - a. Describe the level of detail desired
  - b. Determine the layout of the data returned to the caller
  - c. Are described in the i Information Center
  - d. All of the above

The answer is all of the above.

© Copyright IBM Corporation 2011

## Checkpoint solutions (2 of 2)

IBM i

4. True or False: An 8-character format name must be supplied as a parameter to retrieve APIs.

The answer is true.

5. The \_\_\_\_\_ library on the i system can be used to code the data structure (DS) for the format of the returned data and the prototype for an API.

- a. QSYS
- b. QGPL
- c. QSYSINC
- d. QRPGLE

The answer is QSYSINC.

© Copyright IBM Corporation 2011

## Unit 3

# Checkpoint solutions (1 of 2)

IBM i

1. True or False: Using the **DEFINE** parameter of the **CRTBNDRPG** or the **CRTRPGMOD** command is the same as coding the /DEFINE condition-name directive on the first line of the RPG IV source member.  
The answer is true.
  
2. When testing conditions with directives within a source member:
  - a. /ELSEIF and /ELSE are invalid outside of an /IF group
  - b. An /IF group can contain only one /ELSE directive
  - c. Every /IF must have a matching /ENDIF
  - d. All of the aboveThe answer is all of the above.
  
3. By specifying \_\_\_\_\_ as the data type for a stand a lone field, the field is identified as a pointer field.
  - a. P
  - b. B
  - c. \*
  - d. BASEDThe answer is \*.

© Copyright IBM Corporation 2011

## Checkpoint solutions (2 of 2)

4. The \_\_\_\_\_ keyword is specified for a data structure or stand a lone field in order to create a pointer field with the same name specified as the keyword's parameter.
- BASED
  - P
  - B
  - \*
- The answer is BASED.
5. True or False: Before a based data structure or stand a lone field can be used, the basing pointer must be assigned a valid address.
- The answer is true.
6. In order to use a list API, you must create a \_\_\_\_\_ to hold its output.
- System heap
  - Data area
  - User space
  - Control header
- The answer is user space.

© Copyright IBM Corporation 2011

---

## Unit 4

# Checkpoint solutions

IBM i

1. True or False: ILE CEE APIs are automatically bound to the program that calls them without explicit binding when you create your program object.

The answer is true.

2. Operational descriptors allow:

- a. Omission of passed parameters
- b. Flexibility in parameter passing
- c. A parameter to accept different types of strings
- d. Parameters to be passed in any sequence

The answer is flexibility in parameter passing and a parameter to accept different types of strings.

3. True or False: If you omit the feedback code parameter when you are calling an ILE CEE API, the API fails.

The answer is false.

© Copyright IBM Corporation 2011

## Unit 5

# Checkpoint solutions (1 of 2)

IBM i

1. True or False: A notify object for restart information is required for implementing commitment control in your high-level language programs.

The answer is false.

2. A trigger program:

- a. Can be written in any IBM i high-level language
- b. Is associated with a physical file
- c. Is independent from applications
- d. All of the above

The answer is all of the above.

3. The \_\_\_\_\_ command shows trigger information associated with a data file.

- a. DSPFD
- b. DSPFFD
- c. DSPPGMREF
- d. DSPPFM

The answer is DSPFD.

© Copyright IBM Corporation 2011

## Checkpoint solutions (2 of 2)

IBM i

4. True or False: The application causing a trigger program to fire, waits for the trigger program to complete.

The answer is true.

5. Static embedded SQL statements are complete at compile time while dynamic embedded SQL is completed, compiled, and executed during program execution.

The answer is static embedded SQL statements are complete at compile time while dynamic embedded SQL is completed, compiled, and executed during program execution.

© Copyright IBM Corporation 2011

## Unit 6

### Checkpoint solutions (1 of 2)

IBM i

1. True or False: An ILE module can be composed of one or more procedures or subprocedures.  
The answer is true.
  
2. In order to make data items available outside an individual module:
  - a. Binder language must be used.
  - b. Use the EXPORT keyword.
  - c. Bind the modules by copy.
  - d. Bind modules by reference.The answer is use the EXPORT keyword.
  
3. True or False: An ILE program can be composed of one or many modules.

The answer is true.

© Copyright IBM Corporation 2011

## Checkpoint solutions (2 of 2)

IBM i

4. An ILE RPG module:
  - a. May contain a main procedure consisting of H, F, D, I, C, and O specifications.
  - b. May contain zero or more subprocedures coded with P, D, and C specifications.
  - c. May make subprocedures available outside of the module in which they are contained.
  - d. All of the above.

The answer is all of the above.
5. True or False: In order for an ILE RPG module to be created with only sub procedures, the NOMAIN keyword must be specified in an H-specification.  
The answer is true.
6. True or False: The EXPORT keyword is all that is needed to insure that a subprocedure is available outside of the module in which it is packaged.  
The answer is false.
7. A \_\_\_\_\_ simplifies the process of having to list the modules that are to be bound when you run the CRTPGM, UPDPGM, CRTSRVPGM, or UPDSRVPGM commands.
  - a. Module list
  - b. Binding list
  - c. Binding directory
  - d. Module directory

The answer is binding directory.

© Copyright IBM Corporation 2011

## Unit 7

# Checkpoint solutions

IBM i

1. True or False: A control boundary is the first call stack entry after crossing an activation group (AG) boundary.

The answer is true.

2. During ILE program execution, unhandled errors or exceptions:

- a. Go to the system default error handler.
- b. Cause the program to immediately fail.
- c. Are percolated up the call stack until it handled or it reaches a control boundary.
- d. Behave just like errors in OPM programs.

The answer is are percolated up the call stack until it handled or it reaches a control boundary.

3. True or False: An ILE condition handler is a system provided API that can handle, percolate or promote exceptions in an ILE program.

The answer is false.

© Copyright IBM Corporation 2011

# Bibliography

## ***Reference Material on CD or in IBM i Information Center:***

*Rational Development Studio for i ILE RPG Reference Version 7*

*Rational Development Studio for i ILE RPG Programmer's Guide Version 7*

*IBM i Commitment Control*

*DDS Concepts*

*Physical and Logical Files*

*Display Files*

*Printer Files*

*ILE Concepts*

*ILE Application Development Example*

## ***Web URLs:***

IBM's Internet WEB site:

<http://www.ibm.net>

Internet site for ITSO redbooks:

<http://www.redbooks.ibm.com>

IBM i Information Center:

<http://publib.boulder.ibm.com/iseries/v7r1m0>

Easy/400 Website:

<http://www-922.ibm.com/>

## ***ITSO Redbooks***

SG24-5402      *Who Knew You Could Do That with RPG IV?*

## ***Other Useful Websites***

Certification:

<http://www.ibm.com/certify>

RPG IV Developers' Network:

<http://www.rpgiv.com>





**IBM**  
®