

Who Knew You Could Do That with RPG IV?

Modern RPG for the Modern Programmer

Rich Diedrich

Jim Diephuis

Susan Gantner

Jeff Minette

Jon Paris

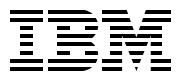
Kody Robinson

Tim Rowe

Paul Tuohy



Power Systems



International Technical Support Organization

**Who Knew You Could Do That with RPG IV?
Modern RPG for the Modern Programmer**

October 2016

Note: Before using this information and the product it supports, read the information in "Notices" on page ix.

Third Edition (October 2016)

This edition applies to Version 7, Release 2, Modification 0, Technology Refresh 1 of IBM i (5770-SS1) and IBM Rational Development Studio for i (5770-WDS).

This document was created or updated on October 20, 2016.

© Copyright International Business Machines Corporation 2016. All rights reserved.

Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Notices	ix
Trademarks	x
IBM Redbooks promotions	xi
Preface	xiii
Authors.....	xiii
Now you can become a published author, too!	xvi
Comments welcome.....	xvi
Stay connected to IBM Redbooks	xvii
Chapter 1. Introduction to RPG IV.....	1
1.1 Why update this book on RPG IV now?	2
1.2 Evolution of the RPG IV language	2
1.3 The future for RPG IV	6
1.4 A road map	7
1.4.1 Step 1: RPG IV and RDi	7
1.4.2 Step 2: Modularization using ILE	8
1.4.3 Step 3: Exploiting database features	8
1.4.4 Step 4: Modernizing the user interface.....	8
1.5 RPG IV sources on the web	9
Chapter 2. Programming RPG IV with style	11
2.1 Just another programming language	12
2.2 RPG is free-form	12
2.2.1 The modern free-form RPG	12
2.2.2 The basics of /Free coding	13
2.2.3 Unsupported op-codes	14
2.2.4 Liberating data and file definitions	15
2.2.5 Data declarations	18
2.2.6 More on data definitions	21
2.2.7 Defining prototypes and subprocedures	23
2.2.8 Miscellaneous topics.....	26
2.2.9 Moving to free-form.....	27
2.3 Modern RPG programs and subprocedures	27
2.4 Naming	28
2.4.1 The names	28
2.4.2 Case	28
2.4.3 Special characters	29
2.4.4 Underscore	29
2.4.5 Named constants	29
2.4.6 Naming conventions	30
2.5 Comments.....	30
2.5.1 Summary comments	30
2.5.2 Detailed commenting	30
2.5.3 Other commenting	31
2.5.4 Positions 1 - 5	31
2.6 Structuring code	31
2.6.1 Declarative code	31

2.6.2 Executable code	32
2.6.3 Multipath comparison	33
2.6.4 Embedded SQL	34
2.7 Using templates and qualified data structures	34
2.8 Qualifying wherever possible	35
2.9 Strings	35
2.10 Subroutines	35
2.11 Older functions	35
2.11.1 RPG built-in indicators	35
2.11.2 Compile-time arrays	36
2.11.3 Multiple occurrence data structures	37
2.11.4 Do not use GOTO, CABxx, or COMP	37
2.11.5 Do not use obsolete IFxx, DOUxx, DOWxx, or WHxx opcodes	37
2.11.6 Use SELECT, WHEN, OTHER, ENDSL for multipath comparison	37
2.12 Embedded SQL	37
2.13 Global definitions	38
2.14 Parameters and prototyping and procedure interfaces	38
2.14.1 Parameters	38
2.14.2 Return value	38
2.14.3 Copy members	38
2.15 The integrated language environment	41
2.15.1 ILE programs	42
2.15.2 Service programs	42
2.15.3 Binding directories	43
2.15.4 Activation groups	43
Chapter 3. Subprocedures	45
3.1 Subprocedure terminology	46
3.1.1 ILE modules	46
3.1.2 Main procedure	46
3.1.3 Built-in functions	46
3.1.4 Subroutines	46
3.2 Advantages of using subprocedures	47
3.3 The anatomy of a subprocedure	48
3.3.1 Subprocedure definition	48
3.3.2 Procedure-interface definitions	50
3.3.3 Order of coding the source elements	50
3.3.4 Calling your subprocedures	52
3.4 Moving from subroutines to subprocedures	53
3.4.1 Why use subprocedures	53
3.4.2 Subroutine example DATESUBR	53
3.4.3 Transforming a subroutine to a subprocedure	55
3.4.4 DATEMAIN1 subprocedure example	57
3.5 Using subprocedures efficiently	59
3.5.1 Using /COPY members for prototypes	59
3.5.2 Using the MAIN control option keyword and a prototype for the main procedure	60
3.5.3 Subprocedures using subprocedures	61
3.5.4 Using an ILE service program	62
3.6 More on subprocedures	66
3.6.1 The power of prototyping	66
3.6.2 Parameter passing styles	70
3.6.3 Using procedure pointer calls	72

Chapter 4. An ILE guide for the RPG programmer	77
4.1 Introduction to ILE	78
4.1.1 Modules and binding	78
4.1.2 Service programs	79
4.1.3 Export and import	79
4.1.4 Binder language source	79
4.1.5 Binding directories	80
4.1.6 Activation groups	80
4.1.7 CL commands used with ILE and RPG.....	81
4.2 ILE tips for the RPG programmer	82
4.2.1 Creating programs from modules (binding by copy).....	82
4.2.2 Binding service programs to programs	101
4.2.3 Service programs, binder language, and signatures.....	103
4.2.4 Using binding directories.....	107
4.2.5 Activation groups	108
4.2.6 Call stack and error handling	121
4.3 Additional CL commands and useful ILE APIs	139
4.3.1 Additional CL commands	139
4.3.2 Some useful APIs to get information on ILE objects.....	139
4.4 More information about ILE and shared open data paths.....	140
Chapter 5. Application programming interfaces	143
5.1 Finding APIs	144
5.2 C functions	145
5.2.1 Prototype mapping example.....	146
5.2.2 A more complex prototype mapping example.....	147
5.3 POSIX interfaces.....	150
5.3.1 Accessing the IFS.....	150
5.4 IBM i system interfaces.....	151
5.4.1 Standard techniques.....	151
5.4.2 List objects (QUSLOBJ) API.....	154
5.5 Creating a reusable API	155
5.5.1 Making a web service	159
5.6 Things to remember	162
Chapter 6. Database access with RPG IV	163
6.1 Externalizing input and output.....	164
6.1.1 What is meant by externalizing	164
6.1.2 Putting theory into practice: An example of externalizing I/O	165
6.1.3 Externalizing example: Overview	166
6.1.4 Externalizing example: Separating database logic from display logic.....	170
6.1.5 Externalizing example: Implementing changes.....	174
6.1.6 Externalizing example: Other possibilities.....	177
6.1.7 Summary.....	178
6.2 Embedded SQL	178
6.2.1 Rules for embedding SQL statements	178
6.2.2 SQL preprocessor	179
6.2.3 Error and exception handling	179
6.2.4 Using a cursor	181
6.2.5 An embedded SQL program example.....	183
6.2.6 Source code for the SQLEMBED program	184
6.3 Stored procedures	188
6.3.1 Creating an external procedure	189
6.3.2 Creating an SQL procedure	189

6.3.3 Starting a stored procedure and returning the completion status	191
6.3.4 A stored procedure example.	191
6.4 DB2 call level interface	200
6.4.1 Differences between DB2 CLI and embedded SQL	201
6.4.2 Writing a DB2 CLI application.	202
6.4.3 Initialization and termination	202
6.4.4 Transaction processing.	204
6.4.5 Diagnostic tests.	207
6.4.6 Data types and data conversion	208
6.4.7 Functions	209
6.4.8 Introduction to a CLI example.	221
6.5 Trigger programs.	236
6.5.1 Adding a trigger program to a file	237
6.5.2 Creating a trigger program	238
6.6 Commitment control	241
6.6.1 File journaling	242
6.6.2 Using commitment control with RPG native file operations	242
6.6.3 Using commitment control with embedded SQL.	244
6.6.4 Using commitment control with the CLI interface	245
6.7 A note about globalization.	246
6.8 More information about database access with RPG IV.	246
Chapter 7. Exception and error handling	249
7.1 Introduction to exception and error handling.	250
7.2 What is an exception/error	250
7.3 Trapping at the program level	251
7.3.1 Program exception/errors	251
7.3.2 The fatal program	253
7.3.3 File exception/errors	254
7.3.4 Can it get any easier.	255
7.4 Trapping at the operation level	255
7.4.1 Error extender	255
7.4.2 Status codes	257
7.4.3 Monitor groups	260
7.4.4 Information data structures	264
7.4.5 ILE condition handlers	266
7.5 Subprocedures and exception/errors	266
7.5.1 Percolation	266
7.5.2 Trapping percolated errors	271
7.5.3 Identifying a percolated message	272
7.6 ILE CEE APIs	287
7.6.1 Condition management APIs	287
7.6.2 Activation group and control flow APIs	289
7.6.3 Further information	298
7.7 Priority of handlers	298
7.8 The embedded SQL problem	299
7.9 Using percolation: try, throw, and catch	302
7.9.1 A traditional approach.	302
7.9.2 Throw	303
7.9.3 Catch.	305
7.9.4 Using throw and catch	307
7.9.5 The problem with throw and catch	308
7.10 Conclusion	309

Chapter 8. Interfacing	311
8.1 Interfacing with Java	312
8.1.1 Java calling RPG IV	312
8.1.2 RPG IV calling Java	314
8.1.3 Things to remember	334
8.2 Python	335
8.2.1 Calling a stored procedure	335
8.3 PHP	338
Chapter 9. IBM Rational Developer for IBM i	339
9.1 Why you should use Rational Developer for IBM i	340
9.1.1 Integrated compile-time error feedback	342
9.1.2 Outlining your program	342
9.1.3 Editing multiple members at once	344
9.1.4 Modern editor capabilities	345
9.1.5 Moving from SEU/PDM to RDi	347
9.2 Built on Eclipse	348
9.2.1 iSphere tools	348
9.3 What is new in Rational Developer for IBM i	352
9.3.1 Code coverage monitor	352
9.3.2 Integrated emulator	354
9.3.3 IBM i Access Client Solutions and Run SQL scripts	355
9.3.4 RPG formatter	357
9.3.5 More flexible outline view	358
9.3.6 Native Mac OS X support	358
9.3.7 Commenting and uncommenting in RPG, CL, and DDS	359
9.3.8 Hyperlink navigation within source member	359
9.3.9 Importing and exporting RDi configurations and push-to-client support	359
9.3.10 Smaller usability enhancements	360
9.4 Using the RDi debugger	360
9.4.1 Starting a debug session using service entry point	360
9.4.2 Starting a debug session using a debug configuration	361
9.4.3 Debug server and preferences	362
9.4.4 RDi debug activities	362
Chapter 10. Modern RPG comparison as viewed from a young developer	365
10.1 Fixed format verses free-format	366
10.2 File-Specs	367
10.3 Tables	368
10.4 Parameters	368
10.5 Key lists	369
10.6 Array and error lookups	370
10.7 Calling programs from a program	371
10.8 Read commands	372
10.9 Evaluates, checks, and scans	373
10.10 Select statements	374
10.11 Substring and concatenate	375
10.12 Go-To commands	376
10.13 Embedded SQL in free-format 101	377
10.14 Built-in functions you need to know	379
Appendix A. Additional material	381
Locating the Web material	381
Using the Web material	381

Related publications	383
IBM Redbooks	383
Other publications	383
Online resources	383
Help from IBM	385

Notices

This information was developed for products and services offered in the US. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing, IBM Corporation, North Castle Drive, MD-NC119, Armonk, NY 10504-1785, US

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation, registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at "Copyright and trademark information" at <http://www.ibm.com/legal/copytrade.shtml>

The following terms are trademarks or registered trademarks of International Business Machines Corporation, and might also be trademarks or registered trademarks in other countries.

DB2®
developerWorks®
IBM®
Integrated Language Environment®

Language Environment®
OS/400®
Rational®
Redbooks®

Redpaper™
Redbooks (logo) ®
RPG/400®
System i®

The following terms are trademarks of other companies:

Evolution, and Inc. device are trademarks or registered trademarks of Kenexa, an IBM Company.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, and the Windows logo are trademarks 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 company, product, or service names may be trademarks or service marks of others.

Find and read thousands of IBM Redbooks publications

- ▶ Search, bookmark, save and organize favorites
- ▶ Get personalized notifications of new content
- ▶ Link to the latest Redbooks blogs and videos

Get the latest version of the **Redbooks Mobile App**



Promote your business in an IBM Redbooks publication

Place a Sponsorship Promotion in an IBM® Redbooks® publication, featuring your business or solution with a link to your web site.

Qualified IBM Business Partners may place a full page promotion in the most popular Redbooks publications. Imagine the power of being seen by users who download millions of Redbooks publications each year!



ibm.com/Redbooks

About Redbooks → Business Partner Programs

THIS PAGE INTENTIONALLY LEFT BLANK

Preface

Application development is a key part of the IBM® i businesses. The IBM i operating system is a very modern robust platform to create and develop applications. The RPG language has been around for a long time, but has not stopped being transformed into a modern business language.

This IBM Redbooks® publication is focused on helping the IBM i development community understand what is modern RPG. The world of application development has been rapidly changing over the past years. The good news is that IBM i has been changing right along with it, and has made significant changes to the RPG language. This book is intended to help developers understand what modern RPG looks like and how to move from older versions of RPG to a newer modern version. Additionally, this book covers the basics of ILE, interfacing with many other languages, and the best tools for doing development on IBM i.

Using modern tools, methodologies, and languages are key to continuing to stay relevant in today's world. Being able to find the right talent for your shop is key to your continued success. Leveraging the guidelines and principles in this book can help set you up to find that talent today and into the future.

This publication is the result of work that was done by IBM, industry experts, business partners, and some of the original authors of the first two editions of this IBM Redbooks publication. Not only is the information important for developers, it is important for the business decision makers (CIO for example) to understand that the IBM i is not an 'old' system. The IBM i has modern languages and tools, it's really just a matter of what you choose to do with the IBM i that defines its age.

Authors

This book was produced by a team of specialists from around the world working with the International Technical Support Organization, Rochester Center.



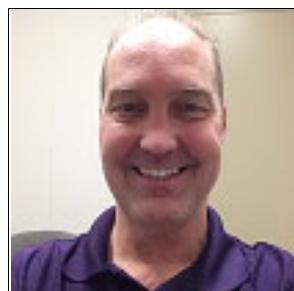
Rich Diedrich is President and IBM i wizard at Rich Diedrich Consulting, LLC. Rich provides consulting and programming assistance on IBM i topics including application modernization, web enablement, encryption, application communication, system interfaces, or anything weird or unusual. Before retiring from IBM to start his own consulting practice, Rich was a Senior Technical Staff Member and Master Inventor in Lab Services at IBM Rochester where he assisted hundreds of IBM i business partners and customers with their applications, wrote articles, presented to IBM i user groups, and was an expert witness for the US Department of Justice during the trial of the programmers who worked for Bernie Madoff. You can reach him by email at rich@richdiedrich.com



Jim Diephuis has been with IBM for over 38 years. He worked in the database area of the S/38 and AS/400 for many years before moving to what is currently the IBM i Lab Services team. Jim is currently the team lead for the application team in Lab Services. He helps clients understand how to modernize their applications through the use of Rational® Developer for i, modern RPG, and general principles of modern application development.



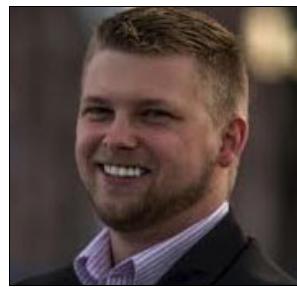
Susan Gantner began her career as a Programmer for companies in Atlanta on various platforms. After working with her first IBM System/38, she vowed to stick with that platform. The System/38 became the IBM AS/400, and then renamed a few times to what we know today as IBM i. After application programming for 10 years outside of IBM, she joined IBM as a System Engineer and later worked in both the IBM Rochester and IBM Toronto labs supporting developers on her favorite platform. Today, post IBM, she works primarily as an educator teaching modern application development techniques through on site classes that are customized for specific companies needs and at technical conferences. She has authored many articles on various topics that are relevant to the IBM i developer.



Jeff Minette has been part of the Lab Services and Training for the past 17 years. Over his consulting career, Jeff has been assisting many customers with modernizing their IBM i applications. This has been through both lab services application modernization workshops and by working directly with development teams as they modernize their applications. Though he has been primarily focused on IBM i native applications and incorporating new technologies, this has expanded through the years to include both client and server side web technologies in both an architectural and developmental role.



Jon Paris has spent most of his 45 years in the IT industry working with midrange systems. His experience covers the spectrum from operations to programming to management. In the early 1980's, Jon was introduced to the IBM System/38 and fell in love. Eventually that love affair led to him joining the IBM Toronto Lab where he worked on the COBOL and RPG compilers for the AS/400. He was also with the initial team that designed the RPG IV compiler. Since leaving IBM, Jon has spent his time helping customers to modernize their applications and development practices. He does this through custom training classes, magazine articles, conference education, web casts, and as one of the author RedBooks such as this one and many others.



Kody Robinson is a developer for Arkansas Electric Cooperative Corporation, located in Little Rock, AR. There he develops on the IBM i platform and uses a variety of languages to deliver applications to his clients. Kody graduated from undergrad in 2014 at the University of Arkansas at Monticello with a bachelors in Computer Information Systems. During this time he worked at a banking institution where he caught his first glimpse of an AS/400. Fast forward to 2016 and not only is he still on the IBM i, but he's leveraging his knowledge to help modernize RPG in his work place, help implement newer solutions, modernize file usage, and more. Kody was the recipient of the 2016 IBM/Common Innovation Award and helps proactively advocate for the platform. Kody is now finishing his masters and hopes to teach part-time at colleges to help fill the world with more RPG programmers.



Tim Rowe is the Business Architect for Application Development and Systems Management for IBM i. He has been working on IBM i for the past twenty plus years in areas that ranged from core operating system to Web middleware. He spends much of his time talking to IBM i customers about modernization. How they can move forward and what IBM i has to offer in order to be successful not just today but for the next decade and beyond.



Paul Tuohy, author of Re-engineering RPG Legacy Applications and The Programmer's Guide to iSeries Navigator, is one of the most prominent consultants and trainer/educators for application modernization and development technologies on the IBM Midrange. He currently holds positions as CEO of ComCon, a consultancy firm based in Dublin, Ireland, and founding partner of System i® Developer, the consortium of top educators who produce the acclaimed RPG & DB2® Summit conference. Previously, he worked as IT Manager for Kodak Ireland Ltd. and Technical Director of Precision Software Ltd. In addition to hosting and speaking at the RPG and DB2 Summit, Paul is an award-winning speaker at COMMON, COMMON Europe Congress, and other conferences throughout the world. His articles frequently appear in iProDeveloper, The Four Hundred Guru, RPG Developer, and other leading publications. Paul also hosts the popular iTalk with Tuohy podcast interviews.

Thanks to the following people for their contributions to this project:

Barbara Morris

Chief Architect for RPG Compiler, Toronto

Edmund Reihhardt

Architect for Rational Developer for i

Debra Landon

IBM Redbooks Project Leader, Rochester, MN

Thanks to the authors of the previous editions of this book.

- ▶ Authors of the first edition, *Who Knew You Could Do That with RPG IV? Modern RPG for the Modern Programmer*, published in February 2000:

Brian R Smith
Martin Barbeau
Susan Gartner
Jon Paris
Zdravko Vincetic
Vladimir Zupka

- ▶ Authors of the second edition, *Who Knew You Could Do That with RPG IV? Modern RPG for the Modern Programmer*, published in April 2000:

Brian R Smith
Martin Barbeau
Susan Gartner
Jon Paris
Zdravko Vincetic

Now you can become a published author, too!

Here's an opportunity to spotlight your skills, grow your career, and become a published author—all at the same time! Join an ITSO residency project and help write a book in your area of expertise, while honing your experience using leading-edge technologies. Your efforts will help to increase product acceptance and customer satisfaction, as you expand your network of technical contacts and relationships. Residencies run from two to six weeks in length, and you can participate either in person or as a remote resident working from your home base.

Find out more about the residency program, browse the residency index, and apply online at:
ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our books to be as helpful as possible. Send us your comments about this book or other IBM Redbooks publications in one of the following ways:

- ▶ Use the online **Contact us** review Redbooks form found at:
ibm.com/redbooks
- ▶ Send your comments in an email to:
redbooks@us.ibm.com
- ▶ Mail your comments to:

IBM Corporation, International Technical Support Organization
Dept. HYTD Mail Station P099

2455 South Road
Poughkeepsie, NY 12601-5400

Stay connected to IBM Redbooks

- ▶ Find us on Facebook:
<http://www.facebook.com/IBMRedbooks>
- ▶ Follow us on Twitter:
<http://twitter.com/ibmredbooks>
- ▶ Look for us on LinkedIn:
<http://www.linkedin.com/groups?home=&gid=2130806>
- ▶ Explore new Redbooks publications, residencies, and workshops with the IBM Redbooks weekly newsletter:
<https://www.redbooks.ibm.com/Redbooks.nsf/subscribe?OpenForm>
- ▶ Stay current on recent Redbooks publications with RSS Feeds:
<http://www.redbooks.ibm.com/rss.html>



Introduction to RPG IV

The RPG language has been around for a very long time. Originally it was a language that was built to work with the punch card. Many of the style and coding restrictions of the language are a result of being restricted to the punch card format. The good news is that the RPG language has been under going a process of transformation. The updates and changes to the language are intended to help make the programmer more productive as well as continue to move the language toward the modern world of programming. Today, RPG is considered a modern business programming language.

One of the great things about RPG is the backward compatibility that is still supported today. Of course, you can make the same argument that this is one of the worst things about the language. The RPG program has never been forced to stop doing ‘old’ things and move forward. It’s always been more the approach of there are great new things you can take advantage of. Today, if you are leveraging the latest in RPG and IBM i technology, you are in great shape for a successful future. The new modern RPG language, is something that will look and feel very natural to the developer of today. While most university students are not taught RPG in school (although there are many schools that do still offer RPG through the IBM Academic Initiative Program) the latest style and tools are very similar to most other languages they have learned. This makes the transition to RPG very smooth.

The RPG IV language was introduced in Version 3 Release 1 (V3R1) of OS/400® and has continued to evolve and mature. Even though RPG IV has been around now for over 21 years and the Integrated Language Environment® (ILE) has been available since Version 2 Release 3 (V2R3), many RPG programmers are still confused by the relationship. Using RPG IV does not necessarily require the use of, or the understanding of, the Integrated Language Environment (ILE). However, using ILE is where RPG IV gets its power. Understanding how to use ILE to minimize rewriting code and maximizing code reuse is what continues to make the RPG language relevant today. For more information about how to take advantage of ILE see Chapter 4, “An ILE guide for the RPG programmer” on page 77.

In addition to the many changes to the actual RPG language it self, there have also been many changes in how applications are structured, how you access data, how data is passed from application to application, and also how you actually go about building applications. Many of these topics are covered to some level in this book. The rest of this chapter helps you to understand what has changed, and how you can go about making the transition to modern RPG.

1.1 Why update this book on RPG IV now?

This publication was first published in February 2000 and updated in April of that year. That is a very long time ago. The world of programming and IBM i development has changed considerably in the past 17 years. Many of the interesting examples that were included in the first book, those are now functions that have been included in to the RPG language its self. In addition, the biggest reason is the transformation of the RPG language from a language that was built and styled for use with a punch card to a very modern business language.

Much of the content of this book discusses the many features that make up modern RPG. From the style of the language, the methodologies used, how you access data, to the tooling you use to build and develop applications in today's modern world. For the modern developer, Rational Developer for IBM i (RDi) is now the development tool of choice. PDM and SEU are no longer being maintained.

The system itself has undergone multiple changes both in hardware and names since the original book was published. It was time to update this document to show people what new magic can be done with this updated language. RPG is still relevant and a key player in the newer environments where the main interface is now a web browser and the system is more about managing a businesses data rather than being the only computer a business might have. However, several of the chapters of the previous version of this book are still relevant to today's programming environment. If you have read the old version, you will still find some of the same wisdom in this updated version with updated examples.

Is RPG IV a dead language? The answer to this question is no. The RPG language continues to evolve and change. The IBM i development team continues to listen to the development community and update RPG with the newest features to help you succeed not just today ,but well into the future.

1.2 Evolution of the RPG IV language

RPG IV was introduced over 21 years ago with Version 3 Release 1 (V3R1) of OS/400. It was a great improvement from the RPG III language with its limited column spaces as shown in Figure 1-1, to a wider columnar structure as shown in Figure 1-2.

```
I          DS
I
I
I          1  5 RES
I          6  80RESN
C          MOVEL 'ABC'    RES
C          MOVE 5      RESN
C          SETON           LR
```

Figure 1-1 RPG III code with limited column spaces

```
D          DS
D  RES          1      5
D  RESN         6      8  0
C          MOVEL     'ABC'    RES
C          MOVE      5      RESN
C          SETON           LR
```

Figure 1-2 RPG IV with wider column structure

RPG IV also introduced data specifications, built-in functions (BiFs), mixed case source, free-form expressions, larger variable names, and other features today's RPG programmer takes for granted.

With each release of the operating system, more features have been added. Over the years each release the RPG language has been changed to 're-invent' it as a modern business language. These enhancements and changes to the language including free-form calculation specifications, prototyped procedures, additional data types, more BiFs, and many other enhancements so the code looked more like what is shown in Figure 1-3.

```
D          DS
D  RES      5
D  RESN     3  0
/free
  res = 'ABC';
  resn = 5;
  *inLR = *on;
  return;
/end-free
```

Figure 1-3 RPG code with the partial free-form

With IBM i V7R1 PTF SI51094 (DB2 PTF group SF99701 level 26 for the SQL precompiler) there is now a completely free-form version of the language as shown in Figure 1-4.

```
dcl-ds resDS;
  res char(5);
  resn packed(3:0);
end-Ds;

resDS.res = 'ABC';
resDS.resn = 5;
*inLR = *on;
return;
```

Figure 1-4 Fully free-form RPG IV

The free-form and older specifications can be intermixed in the code without the need for the '/free' and '/end-free'. The code must still appear between columns 8 and 80 of the source member. With the release of the latest RPG updates, the 8-80 column restriction has been removed. You can specify '**FREE' in the first column and from that point on, there are no column restrictions. The right hand side can go out as wide as you want. The only restriction with this is you cannot mix old and new styles of RPG. This only supports fully free-form RPG, see "Modern RPG code" on page 5 for details.

Note: All of the examples shown in this book are given in the this free-form version. This form comes in the initial release of IBM i V7R2.

Figure 1-5 shows a version of the same program shown in Figure 1-4 on page 3 with a main procedure defined and utilizing a template to define the data structure used in the routine.

```

ctl-opt main(simpleExample);
ctl-opt copyright('(C) Copyright IBM Corp. 2015. All rights reserved.');

// Template for common data structure
dcl-ds resT qualified template;
  res  char(5);
  resn packed(3:0);
end-Ds;

// -----
// Prototype for procedure: simpleExample
// -----
DCL-PR simpleExample extpgm('MYEXAMPLE');
END-PR ;

dcl-proc simpleExample;
  DCL-PI simpleExample;
  END-PI ;

  dcl-ds resDS likeds(resT);

  resDS.res = 'ABC';
  resDS.resn = 5;

  return;

end-Proc;

```

Figure 1-5 Main procedure utilizing a template to define the data structure used in the routine

The 'main()' in Figure 1-5 tells the RPG compiler that it does not need to include the RPG-cycle in the machine level code (note that the RPG-cycle should not be used any longer). The 'template' keyword makes the variable resT a model that can be used to define other variables, like resDS in the procedure.

Now lets turn the program into a procedure that sets the values of the data structure. For this, the template and prototype are put into a separate source member. See Figure 1-6.

```

// Copy source for a simple example.

// Template for common data structure
dcl-ds resT qualified template;
  res  char(5);
  resn packed(3:0);
end-Ds;

// -----
// Prototype for procedure: simpleExample
// -----
DCL-PR simpleExample;
  resDS  likeds(resT);
END-PR ;

```

Figure 1-6 Procedure that sets the values of the data structure

Then, the source is changed to copy in the prototype, change the parameter list to match, and change the whole thing to not be a main procedure. See Figure 1-7.

```
ctl-opt nomain;
ctl-opt copyright('(C) Copyright IBM Corp. 2015. All rights reserved.');

// Prototype for data structure and procedure
/copy gcpysrc,myexample

dcl-proc simpleExample;

DCL-PI simpleExample;
    resDS      likeds(resT);
END-PI ;

resDS.res = 'ABC';
resDS.resn = 5;

return;

end-Proc;
```

Figure 1-7 Changing the source to copy in the prototype

The latest enhancements to the language allow the code to be totally free, where the code can start in column one and go for as long as the source member is defined (minus the 12 bytes for sequence number and last changed date). This is part of the enhancements in V7R1 PTF SI58136 (PTF SF99701 level 38 for the SQL precompiler) and V7R2 PTFs SI58137 and SI58110 (PTF SF99702 level 9 for the SQL precompiler).

Figure 1-8 shows what the code shown in Figure 1-7 could look like in the future.

```
**free
ctl-opt nomain;
ctl-opt copyright('(C) Copyright IBM Corp. 2015. All rights reserved.');

// Prototype for data structure and procedure
/copy gcpysrc,myexample

dcl-proc simpleExample;

DCL-PI simpleExample;
    resDS      likeds(resT);
END-PI ;

resDS.res = 'ABC';
resDS.resn = 5;

return;

end-Proc;
```

Figure 1-8 Modern RPG code

The '**free' in the first line of the code in Figure 1-7 tells the compiler there are no column limitations or old code specifications in this source member. The code in the /copy member could still be defined within column limitations or even with statement type specifications. Using this version of RPG IV makes it easier to store your source in the integrated file system (IFS) or a stream oriented change management system with fewer concerns about white space.

Examples in this book are based on IBM i V7R2

The RPG programs and IBM i database libraries used in this book are available for you to download from the Internet. These examples were developed using an IBM i system and the ILE RPG compiler (5770-WDS, option 31) at IBM i version 7.2. See Appendix A, “Additional material” on page 381 for instructions on how to download the IBM i save file containing the RPG examples used in this book.

Note: Some of the programming examples in this book use functions and features of the compiler and IBM i system that will only compile and run on IBM i V7R2. Others might run on a V7R1 system without modifications. The full free-form RPG for almost all statement types was not available before V7R1 technology refresh (TR) 7, therefore most of the examples will not run on any system prior to that without modification.

1.3 The future for RPG IV

As the example in Figure 1-8 on page 5 shows, there has been significant investment made to enhance the RPG IV language since its inception. That trend is planned to continue into the foreseeable future. A long and significant list of potential enhancements are in the works by the compiler developers for future releases and versions of the RPG IV compiler.

One significant area where changes are likely to continue, relate to making it easier to integrate RPG with newer interfaces. XML is already easier to do with RPG. Several of these new interfaces are examined in Chapter 8, “Interfacing” on page 311. The ability to integrate the RPG code that has been reliably serving our business application requirements for years is critical to the smooth implementation of these new technologies.

While it is quite possible to call between RPG and Java applications today (for example, using the program call from the AS/400 Toolbox for Java or the JNI support in V4R4), there is still much room for improvement in making the integrated support between the languages much easier and more robust. Enhancements in this area are important to the future utilization of both languages and are a high priority.

It is also important to continue the growth of RPG as a language in its own right. With as many dramatic enhancements as seen in the past releases, there is still room to continue to evolve the language as programmers evolve in their use of the language.

The RPG development team in the IBM Toronto laboratory is keenly aware of the need for and the advantage of listening to the RPG programming community to help steer the direction of their language of choice. RPG programmers will see even more dramatic enhancements in both the near term and long term that will enhance their productivity and their programming style options in RPG. Many of these enhancements have been guided by input from RPG programmers around the world.

A future for RPG programmers

The RPG language has a long and bright future ahead. How can today’s RPG programmers position themselves, their company, and their applications to take full advantage of what RPG has to offer today and tomorrow? Reading this book is a good start. Wherever you happen to be on the scale of taking advantage of the modern RPG IV language, this book has something to offer.

For example, if you have not yet moved your skills or your applications to RPG IV, you will find information in this book to help you see the advantages of doing so (or justify the move to your management). For those already using RPG IV and perhaps even using the more

advanced features, such as subprocedures and using C functions via RPG prototypes, you will find some examples of exploiting these features further and options to improve your coding style.

1.4 A road map

It might be helpful to suggest a road map for programmers looking to move from pre-RPG IV environments to taking advantage of today's RPG language. The following steps are discussed in this section:

- ▶ 1.4.1, "Step 1: RPG IV and RDi" on page 7
- ▶ 1.4.2, "Step 2: Modularization using ILE" on page 8
- ▶ 1.4.3, "Step 3: Exploiting database features" on page 8
- ▶ 1.4.4, "Step 4: Modernizing the user interface" on page 8

1.4.1 Step 1: RPG IV and RDi

The first step for those programmers whose skill and applications are not yet moved to RPG IV is clear. If your applications and programming skill are still primarily or exclusively in RPG/400® (or RPG III, as it is sometimes called), it is long past the time to move forward. Learn RPG IV and move your applications to the new language as soon as possible.

Besides moving to RPG IV, it is time to move off of the 5250 display and PDM/SEU to a modern Integrated Development Environment (IDE). For the IBM i developer, this is Rational Developer for IBM i (Rdi).

Note: All of the example code used in this book has been developed using Rdi.

To enhance your skills in RPG IV, there are many books, publications, classes offered by IBM and others, tutorials, and conferences that provide help on how to take advantage of the latest technologies while using RPG IV.

Once in RPG IV and throughout your journey, adopt a consistent style of coding in RPG IV. The style guide included in this book (see Chapter 2, "Programming RPG IV with style" on page 11) provides a good place to start in developing your style for RPG IV coding. While it is typically not feasible to rewrite all your existing code to meet the requirements of your style guide, certainly new code and enhancements can use the style features that you decide work best for your environment. Some of the conversion tools available on the market also help in making some basic style adjustments, such as the use of upper and lower case, the use of the dcl-s specification for stand-alone work fields, and using free-form operations.

It is important to avoid the pitfall of using RPG IV in the same style as you have used with previous RPG languages. Keep up with the latest techniques and enhancements of the compiler and continue to enhance your style and coding standards to fully exploit all that this language has to offer.

Convert all fixed format RPG to the new totally free-format RPG. There are several tools in the industry that do a great job doing this sort of transformation:

- ▶ ARCAD Transformer, the IBM i strategic partner support, this can be purchased either directly from ARCAD or though the IBM software ordering system:

<http://arcadsoftware.com/products/arcad-transformer-ibm-i-refactoring-tools/>

- ▶ Linoma Software RPG Toolbox:
<http://www.linomasoftware.com/products/rpg-toolbox>

1.4.2 Step 2: Modularization using ILE

After moving your applications to RPG IV, begin exploring the potential use of ILE. ILE enhances your ability to write in a modular style and is required for many of the advanced features of the RPG IV language. ILE service programs provide the foundation for accessing most of the more modern system functions. It is also the foundation of integrating with Java.

This book helps you to understand many of the basic ILE concepts and the features of RPG IV that require the use of ILE (and their advantages). In addition, there are some other books, manuals, classes, articles, seminars, and conferences that can help you understand how to implement ILE in your applications.

Look for opportunities to modularized your application code. Some examples of ways to do this are included in the various chapters in this book. Modularization, when done well, usually results in higher productivity for programmers and more reliable application logic. Many developers have also experienced improvements in application performance because they could tune a procedure that is now written once and used many times.

1.4.3 Step 3: Exploiting database features

Another step to consider is to look for opportunities to include database features to either replace or add reliability to parts of your application logic. This step might come before, after, or even simultaneously with the ILE implementation step mentioned in 1.4.2, “Step 2: Modularization using ILE” on page 8.

Referential integrity constraints, for example, can be implemented in the database more reliably than in application logic, which depends on all programmers implementing the logic correctly and consistently. Likewise, application logic could be replaced or made more robust by adding check constraints to database fields to implement functions such as range or value list checking. Using SQL instead of native I/O to return the set of data the program really wants can reduce the size of the code as well as improve the performance of the application.

1.4.4 Step 4: Modernizing the user interface

Most businesses today are looking for improved usability and new functions by using a web-enabled user interface. These can be provided with web services that typically require changes to the existing code. Making the code more modular is the first step in providing these services. There are other interfaces that might be required as well. Using a common interface based on XML is a popular theme in new application designs for example.

Some of the current interfaces and how to use them with RPG IV is discussed in Chapter 8, “Interfacing” on page 311.

1.5 RPG IV sources on the web

To find out more about RPG IV, refer to the following general IBM i resources:

- ▶ The IBM i Knowledge Center is your gateway to IBM i technical information for multiple releases and IBM i reference material. You can find this information at:

http://www-01.ibm.com/support/knowledgecenter/ssw_ibm_i/welcome

- ▶ Education on RPG and other IBM i classes can be found on the IBM Training and Skills website:

<http://www-304.ibm.com/services/learning/ites.wss/zz/en?pageType=page&c=a001103>

- ▶ Interaction with other RPG developers can be found on the IBM developerWorks® RPG Cafe:

<https://www.ibm.com/developerworks/community/groups/service/html/communityview?communityUuid=b542d3ac-0785-4b6f-8e53-f72051460822>

For non-IBM general IBM i resources:

- ▶ A good IBM i system specific search site can be found at:
<http://www.search400.com>
 - ▶ *Programming in ILE RPG*, by Brian Meyers and Jim Buck and *Free-Format RPG IV*, by Jim Martin can be found at:
<http://www.mc-store.com>
 - ▶ The Midrange.com website provides a large number of resources for users and developers of IBM i:
<http://www.midrange.com>
- Here, too, you can subscribe to an RPG400-L mailing list. This allows programmers to discuss the main application development language of the IBM i.
- ▶ The IBM i user group called COMMON is located on the Web at:
<http://www.common.org>

Note: This is the USA COMMON website. Click on **Friends** to select your geography for other world-wide COMMON organizations. More directly, if you are interested in European COMMON activity, you can go directly to the COMMON Europe website:

<http://www.comeur.org>

- ▶ IBM i development training, education, articles, and general useful information on modern development on IBM i. The home of the RPG and DB2 Summit conference.
 - System i Developer:
<http://systemideveloper.com/>



Programming RPG IV with style

This chapter provides a style guide to coding RPG programs when using modern RPG which includes fully free-form RPG.

Another source for style guidelines can be found in Appendix D of *Programming in ILE RPG, 5th Edition*:

http://www.qrpglesrc.com/downloads/ILE_RPG_Style_Guide.pdf

When trying to develop guidelines and standards, one of the major challenges is determining what is a standard and what is a guideline. For example, code indentation is a *standard*, but whether the code is indented by two, three, or four characters is a *guideline*. One of the objectives of standards is to ensure that required coding standards are implemented without impeding the creativity of the programmer.

The following topics are discussed in this chapter:

- ▶ 2.1, “Just another programming language” on page 12
- ▶ 2.2, “RPG is free-form” on page 12
- ▶ 2.3, “Modern RPG programs and subprocedures” on page 27
- ▶ 2.4, “Naming” on page 28
- ▶ 2.5, “Comments” on page 30
- ▶ 2.6, “Structuring code” on page 31
- ▶ 2.7, “Using templates and qualified data structures” on page 34
- ▶ 2.8, “Qualifying wherever possible” on page 35
- ▶ 2.9, “Strings” on page 35
- ▶ 2.10, “Subroutines” on page 35
- ▶ 2.11, “Older functions” on page 35
- ▶ 2.12, “Embedded SQL” on page 37
- ▶ 2.13, “Global definitions” on page 38
- ▶ 2.14, “Parameters and prototyping and procedure interfaces” on page 38
- ▶ 2.15, “The integrated language environment” on page 41

2.1 Just another programming language

With the addition of fully free-form RPG, RPG has achieved the distinction of being just another programming language, which means that many of the style guidelines and standards that apply to other programming languages (such as Java and PHP) now apply to RPG. If you are developing in a multi-language environment, try to apply consistent standards to all languages. For example, if the guideline for PHP is to indent code by three characters, the guideline for indenting in RPG should be the same.

Even if you are not developing in a multi-language environment, you can still learn from other languages. For example, use /INCLUDE instead of /COPY and use all uppercase for name constants, as these are conventions that are common to most (if not all) programming languages.

IBM Rational Developer for i can help with the implementation of standards. Learn to use automatic indent, the outline view, content assist, templates, and snippets. All of these features can help automate standards and styles. For more information on IBM Rational Developer for i, see Chapter 9, “IBM Rational Developer for IBM i” on page 339.

2.2 RPG is free-form

Modern RPG programs should contain only full free-form code. They should not be a mixture of fixed-format and/or extended factor-2 and/or free-form. Complete free-form means that a lot of old “bad habits” (MOVE, MOVEL, GOTO, and so on) are no longer available. Coding only free-form code means that a programmer will not accidentally fall back onto old habits and coding practices.

If you decide that existing programs are not going to be converted to free-form RPG, then any modifications (to the programs) should still be made in free-form only.

Whether writing new programs or modifying existing programs, only code in free-form RPG. However, avoid creating the situation where the code switches frequently between fixed-form and free-form code. If you are making several minor changes to a section of fixed-form code, consider changing that section of fixed-form code to free-form before making your changes.

The following topics are covered in this section:

- ▶ 2.2.1, “The modern free-form RPG” on page 12
- ▶ 2.2.2, “The basics of /Free coding” on page 13
- ▶ 2.2.3, “Unsupported op-codes” on page 14
- ▶ 2.2.4, “Liberating data and file definitions” on page 15
- ▶ 2.2.5, “Data declarations” on page 18
- ▶ 2.2.6, “More on data definitions” on page 21
- ▶ 2.2.7, “Defining prototypes and subprocedures” on page 23
- ▶ 2.2.8, “Miscellaneous topics” on page 26
- ▶ 2.2.9, “Moving to free-form” on page 27

2.2.1 The modern free-form RPG

Since the original version of this book was written, much has happened to RPG. It is no longer a column-oriented language that is rooted in punch cards, RPG has become a modern business-oriented language.

The first thing to understand about RPG free-form support is that there are actually multiple types of free-form. Three types are available, if you don't count the original free-form EVAL type expression support that was available in the initial release of RPG IV. As a result you often have to qualify exactly which type you are talking about.

This section describes these different types of RPG free-form in the sequence in which they were released.

First steps

RPG was partially liberated in Version 5.1 and this version is often referred to as */Free RPG*. This is the version that many people think of when the term free-form RPG is mentioned. Here are the main features of this initial release:

- ▶ The introduction of free-form versions of all supported op-codes. For more information, see 2.2.3, "Unsupported op-codes" on page 14.
- ▶ The introduction of the semicolon (;) as a statement terminator.
- ▶ Operand names were no longer constrained to 14 characters to fit within boxes.
- ▶ The EVAL and CALLP op-codes are optional in most cases.
- ▶ A number of op-codes and other features such as resulting indicators and result field definition were removed.
- ▶ A number of new built-in functions (BIFs) were added, which replaced some existing op-codes and provided improved functions.
- ▶ By coding the compiler directive /Free, subsequent RPG logic could be placed anywhere within columns 8 - 80. There was no need for a "C" in position 6.
- ▶ A new style of commenting was introduced. The use of a double / allowed for the addition of comments to the end of calc specs and could be used for full-line comments.

2.2.2 The basics of */Free* coding

Many people are confused about how to code often used op-codes such as CHAIN, EXFMT, and other op-codes in free-form, but it is simple. All of the free-form op-codes follow the same basic rule. The sequence is *opcode* followed by the optional extender, then *factor1*, followed by *factor2*. In cases such as data structure (DS) I/O operations, where the result field is not necessarily used to hold a result, but is an additional factor, then it follows *factor2*.

In the following example, key chain filename, simply becomes chain key file name; it is that easy.

A small example

As you can see in Example 2-1 on page 14, every single statement is followed by a semicolon. This is true in all cases with no exceptions. In this way, RPG differs from some languages, such as C and PHP, where a semicolon is used in most cases but not all of them.

For example, an alternative character such as a { is used in conditionals, such as IF statements and certain other constructs, such as function definitions. The RPG rule is much simpler to remember, you must always have a semicolon at the end of every line.

Operations can extend across as many lines as you want with no need for special continuation characters, unless you are extending a literal. But, you *cannot* have multiple operations on a single line.

You can indent the code as you want, which aids greatly in readability, especially where nested IFs and other statements are used.

Example 2-1 /Free coding

```
/Free
  Read(E) TransFile;           // Read file to prime do loop

  DoW not %EOF(TransFile);    // Continue until all processed
  If %Error;
    Dsply 'The read failed';
    Leave;
  Else;
    Chain(N) CustNo CustMast CustData;
    CustName = %Xlate(Upper : Lower : CustName);
    ExSr CalcDividend;
    Read(E) TransFile;
  EndIf;
EndDo;

BegSr CalcDividend;
  TotalSales = %XFoot(MthSales);
  Eval(H) Dividend = TotalSales / 100 * DivPerc;
  Record_transaction();
EndSr;
```

The example shown in Example 2-1 does not use many of the new features other than the re-sequencing of operators in statements such as the CHAIN. It does use indentation and features such as the built-in function %EOF. %EOF, and other I/O related BIFs have been available for some time, but their use was optional because resulting indicators could still be used in previous versions of RPG.

Within the subroutine, the first line is an EVAL, but the ability to omit the opcode was used. However, the second line required that the EVAL be coded because the operation extender (H) for half adjust was required. The final line of the subroutine is also missing an opcode in this example (it is CALLP). The compiler can easily differentiate between EVAL and CALLP because EVALs always have an = immediately following the first operand.

Another feature that is demonstrated by this example is the use of the new // style comment lines, which can be used at the end of the line. These lines can be useful in documenting operations without needing an additional line. It also makes it obvious that the comment relates to the line rather than the block of code that follows it.

2.2.3 Unsupported op-codes

Not all of the fixed-format op-codes are supported in free-form. Indeed, many of them are unsupported ones, such as GOTO, that were included only for compatibility reasons.

The unsupported opcodes fall into five basic categories:

- ▶ Opcodes whose use in present day programming is discouraged.

These opcodes include anything that uses resulting indicators, such as COMPxx, because there are no resulting indicators in the free-form calculation specifications. A similar situation applies to the DEFINE op-code because there is no length definition capability.

The GOTO op-code also was removed.

- ▶ Op-codes that were already obsolete because of improved support through expressions were also dropped, including DOxyy, GOTO, IFxx, ANDxx, and ORxx.
- ▶ Op-codes for which new support is being provided, often in the form of new built-in functions, such as %XFOOT.
- ▶ Op-codes that have existing but not identical expression support. For example, ADD is replaced by +, SUB by -, and so on.
- ▶ Op-codes that were replaced by a combination of new and existing support. An example is the date operations ADDDUR and SUBDUR, which are replaced by a combination of the new BIFs, such as %Days, %Months, and %Years together with simple + and - expression operators.
- ▶ Op-codes that are no longer supported.

Of these, the one that was the most controversial was MOVE. The major reason for its removal was that when you look at a MOVE operation you can tell almost nothing about what is happening unless you know the full definitions of both factors. All you really know is that some data is moved from the factor 2 field to the result field. But, without the full definitions, you do not know whether any of the following items are true:

- The date was simply copied byte for byte.
- The type of the data was changed (was converted from character to numeric).
- The data was truncated.
- Two strings were effectively concatenated.
- High-order numeric digits were discarded without warning.

Because of these and other issues, MOVE operations often caused misunderstandings and errors. IBM replaced them with EVAL type operations where the intent of the code was more obvious.

2.2.4 Liberating data and file definitions

The next significant change in free-form support came in IBM i version 7.2, and was retrofitted to version 7.1. This enhancement provided free-form alternatives for all data declarations and for many file declarations, replacing the old F and D specs. This means that no free-form support is provided for cycle-related files, but only for programmer full procedural files. Free-form alternatives were also provided for the H and P specs.

This release also removed the need for the /Free and /End-Free directives. The compiler now assumes that the source line is in free-form if columns 6 and 7 are left blank.

F and D spec entries have migrated to declarations. A declaration is introduced by the characters DCL and then is followed by an indication of the type of declaration. For example, files are declared by DCL-F, data structures by DCL-DS, and stand-alone fields by DCL-S.

Another change was the introduction of a number of sensible defaults. For example, all files are assumed to be described externally unless otherwise specified. Similarly, all numeric fields are assumed to have zero decimal positions unless otherwise specified. The various defaults are described in the descriptions of the different declarations.

All of the new declarations follow the same basic format, thus you should have no trouble converting to this new way of coding:

- ▶ The DCL-xx comes first.
- ▶ DCL-xx is followed by the name of the item (file, DS, field, and so on).

- ▶ Next come keywords to replace the column-sensitive entries.
- ▶ These keywords are followed by any keywords that are in the D and F spec option columns.
- ▶ The definition is terminated by a semi-colon (;).

Example 2-2 shows how this format works for D-specs.

Example 2-2 D-spec format

```
// D-spec version
d employeeDS      ds
d   firstName          16a  Inz('James')
d   lastname           30a  Inz('Joyce')
d   salary              7p 2 Inz(12500)

// Free-form declaratives version
dcl-ds employeeDS;
  firstName  char(16)    Inz('James');
  lastname   char(30)    Inz('Joyce');
  salary     packed(7:2)  Inz(12500);
end-ds;
```

Keywords such as char and packed replace the column sensitive data type definitions. Unlike the old method, where you might have omitted the data types (a and p), there are no such defaults in data declarations. In the current version, it is necessary to close explicitly a DS with an end-ds.

Declaring files

You do not have to tell the compiler that files are externally described. However, RPG does permit you to specify *EXT as a parameter to the device type keyword. For example, DISK(*EXT).

These new defaults mean that you can omit the device type and the compiler assumes that you are referring to an input DISK file. Similarly, you do not have to specify that a PRINTER file is used for output, it is the default.

The default file usage (input, output, or both) is based on the device type. You specify it only if you want a usage other than the default value.

Therefore, if you are using an externally described and non-keyed database file for input only, the following declarations have the same effect:

```
Dcl-F Orders;
Dcl-F Orders Disk;
Dcl-F Orders Disk(*Ext) Usage(*Input);
```

Table 2-1 shows the defaults for the different file types.

Table 2-1 File type default types

File type	Default usage
DISK (Default type)	Input
PRINTER	Output
WORKSTN	Input/Output
SPECIAL	Input
SEQ	Input

Sensible defaults also are used in cases where the default usage is not sufficient for your task. The compiler takes an incremental approach. For example, to perform updates, the file must be defined as input capable. However, you do not have to specify that definition, the compiler adds *INPUT capability whenever *UPDATE is specified.

Table 2-2 shows the defaults that are used.

Table 2-2 Default file type usage

Usage specified	Additional usage included	Can also be coded as
*Input	None	
*Output	None	
*Update	*Input	*Input: *Update
*Delete	*Update (and therefore *Input)	*Input: *Update: *Delete

Previously, if you specified a "U" to enable a file for update, it was also enabled for delete. As you can see from Table 2-2, this situation is no longer the case in free-form. Specifying *DELETE implies *UPDATE, but the reverse is not true.

Similarly, in order for records to be added to a file, *OUTPUT must now be specified. Previously, file addition was specified by coding an "A" in a specific column, which resulted in a situation where what appeared to be an input-only file could have new records written to it. This situation confused a number of RPG programmers who failed to notice the "A" in the F-spec.

Declaring program-described files

Although most RPG programmers have adopted externally described disk and display files, external described printer files have not been adopted as thoroughly. Perhaps the fact that O-specs remain in fixed format, and in all probability will forever remain that way, will encourage more programmers to make the switch.

Even if you ignore PRINTER files, there are still a few occasions where you may need to program describe a file, for example when writing a generic file processing program.

To define a file as program-described requires that you add a length parameter to the device keyword, for example, **PRINTER(132)**. Similarly, if you must program-describe a keyed disk file, add a key type and length parameter to the **KEYED** keyword: For example:

```
KEYED( *Char : 5 )
```

2.2.5 Data declarations

All declarations begin with DCL and are followed by characters to represent the type of declaration. D-specs are replaced by the DCLs that are shown in Table 2-1.

Table 2-3 Data declarations

Data Declarations	Definition
DCL-S	Defines a stand-alone field.
DCL-DS	Defines the start of a data structure.
DCL-C	Defines a named constant.
DCL-SUBF	Define a DS subfield. Only required if field name matches an RPG op-code name.
END-DS	End data structure. Name of the DS is optional

Data structures must be terminated by an END-DS directive because they are no longer terminated by the beginning of another definition.

Using keywords instead of codes

Everything in the definition is keyword-based, including the data type. For example, fixed length alpha-numeric fields are now declared as CHAR(1en) and packed decimal fields are declared as PACKED(digits : decimals).

There are many benefits to these new formats:

- ▶ The format of the date and time fields is now part of the data type declaration rather than requiring a separate keyword. To define a date field with the USA format, you code DATE(*USA).
- ▶ You no longer must specify the number of decimal positions if it is zero. This is efficient when defining integers, where specifying decimal places never made any sense.
- ▶ The data type for an item must be defined, it no longer defaults. There is no more confusion about whether a field is packed or zoned, it always has the type that you specify.

Table 2-4 lists the fixed-form data types with their corresponding free-form keywords.

Table 2-4 Data definition conversion chart

Fixed form	Free-form	Data type
B	BINDEC	Old style “binary”. Use INT or UNS instead
A	CHAR	Fixed length character
D	DATE	Date field
F	FLOAT	Floating Point
G	GRAPH	Double-byte Graphic Data
N	IND	Indicator
I	INT	Integer
O	OBJECT	Object (Java)
P	PACKED	Packed Decimal

Fixed form	Free-form	Data type
*	POINTER	Pointer
* + PROCPTR option	POINTER(PROC)	Procedure Pointer
C	UCS2	UCS-2
U	UNS	Unsigned Integer
T	TIME	Time
Z	TIMESTAMP	Timestamp
A + VARYING option	VARCHAR	Varying length character
G + VARYING option	VARGRAPH	Varying length graphic
C + VARYING option	VARUCS2	Varying length UCS-2
S	ZONED	Zoned numeric

Examples of free-form declarations

The following are a few stand-alone field definitions to give you an idea of how they are coded:

```
Dcl-S CustName Char(10);
Dcl-S AcctBalance Packed(5:2);
Dcl-S Count Packed(3);
Dcl-S AnotherCount Int(3);
Dcl-S OrderDate Date(*MDY);
```

When it comes to defining data structures, you specify only the DCL at the start of the DS. There is a DCL for subfields as well (DCL-SUBF) but, like EVAL and CALLP, that opcode is optional in most cases. The only time that you must specify it is if the name of the field you are defining is the same as an opcode, such as SELECT or READ.

A simple data structure might look something like the following example:

```
Dcl-DS Address;
  Street Char(30);
  City Char(20);
  State Char(2);
  ZipCode Zoned(9);
    Zip Zoned(5) Overlay(ZipCode);
    ZipPlus Zoned(4) Overlay(ZipCode:5);
End-DS Address;
```

This example demonstrates that the new approach allows overlay fields to be indented to show visually the structure.

Regarding overlays, one difference in the D spec rules is that the **OVERLAY** keyword can no longer reference the DS name. Instead, use the new **POS** keyword to specify the exact position at which the subfield starts.

Here is an example of a fixed-form version of a D-spec technique that can be used in place of compile-time array data. It demonstrates the use of the **OVERLAY** keyword on a DS name.

```
d daysOfWeek      DS
d                               9a  Inz('Monday')
d                               9a  Inz('Tuesday')
d                               9a  Inz('Wednesday')
d                               9a  Inz('Thursday')
d                               9a  Inz('Friday')
d                               9a  Inz('Saturday')
d                               9a  Inz('Sunday')

// Redefine name values as an array
d dayNames                  9a  Overlay(daysOfWeek)
d                           Dim(7)
```

This free-form version shows how the changes are needed. Instead of coding **Overlay(daysOfWeek)**, you **Pos(1)** to position the overlaying array.

In the original version, the field names were left blank. In the new version, you must use the marker “*n” to indicate to the compiler that the field is unnamed because it can no longer determine that from the empty column:

```
dcl-ds  daysOfWeek;
*n char(9) Inz('Monday');
*n char(9) Inz('Tuesday');
*n char(9) Inz('Wednesday');
*n char(9) Inz('Thursday');
*n char(9) Inz('Friday');
*n char(9) Inz('Saturday');
*n char(9) Inz('Sunday');

// Redefine name values as an array
dayNames  char(9) Pos(1) Dim(7);
end-ds;
```

This new **POS** keyword also provides a much “cleaner” approach when defining fields that are in fixed column positions, for example, components of the PSDS or numbered indicators in an INDDS. You can see this approach in the following example, which demonstrates how to associate names with the conventional *INnn indicators. As you can see, adding the indicator number to the end of the name makes it easier to associate the numbers that are used in the DDS with the names that are used in the program.

Here is the fixed-form version:

```
D DspInd          DS           Based(pIndicators)
// Response indicators
D   Exit_03        N  Overlay(DspInd: 3)
D   Return_12       N  Overlay(DspInd: 12)

// Conditioning indicators
D   Error_31        N  Overlay(DspInd: 31)
D   StDtErr_32       N  Overlay(DspInd: 32)
D   EndDtErr_33      N  Overlay(DspInd: 33)

D pIndicators     S           *  Inz(%Addr(*In))
```

Here is the free-form version:

```
dcl-ds DspInd Based(pIndicators);
// Response indicators
  Exit_03      Ind Pos(3);
  Return_12     Ind Pos(12);
// Conditioning indicators
  Error_31      Ind Pos(31);
  StDateError_32 Ind Pos(32);
  EndDateError_33 Ind Pos(33);
end-ds;

dcl-s pIndicators Pointer Inz(%Addr(*In));
```

The new version is a lot “cleaner” because there is no need to specify repeatedly the DS name. As a result, the association to the actual indicator number is clear.

2.2.6 More on data definitions

In the new free-form support, IBM has relaxed some of the rules concerning the use of named constants. This provides some interesting options when defining data.

For example, previously in order to ensure that all fields of a specific type (currency for instance) used a common definition, you could define a basic item definition, perhaps using the TEMPLATE keyword, and then use the LIKE keyword. This was useful but you could not change the data type of the cloned field. If your base definition was packed then all of its clones were also going to be packed.

The relaxed constant support offers an alternative approach since one place where you can now use them is in the length definitions for fields. Take a look at the following example:

```
dcl-c DIGITS 7;
dcl-c DECIMALS 2;

dcl-s customerBalance Packed( DIGITS: DECIMALS);
dcl-s invoiceTotal    Zoned( DIGITS: DECIMALS);
```

Notice that rather than clone the field definition with the LIKE keyword, you can now specify the data type you want while still using a common set of length definitions. Should you need to change the size of these types of field at any time in the future you can do so simply by changing (say) DIGITS from 7 to 9.

The two techniques can also be mixed. For example:

```
dcl-c DIGITS 7;
dcl-c DECIMALS 2;

dcl-s packedCurrency Packed( DIGITS: DECIMALS) Template;
dcl-s zonedCurrency  Zoned( DIGITS: DECIMALS) Template;

dcl-s customerBalance Like(packedCurrency);
dcl-s invoiceTotal    Like(zonedCurrency);
```

On the subject of the LIKE keyword, there is also one small aspect of moving that to a free-form declaration that might not be immediately obvious. For example if you currently have a definition like this:

```
D reportTotal      S           +2    Like(customerBalance)
```

Then to perform similar adjustment to the length of the cloned field in declarations you must code a second parameter, as in the following:

```
dcl-S reportTotal2      Like(customerBalance: +2);
```

Potential pitfalls

This new flexibility in the use of literals does have one potential pitfall. Some keywords that previously accepted literals without the use of quotation marks (such as DTAARA, EXTNAMES, and EXTFLD) now need quotes.

In free-form declarations, an unquoted name always refers to a field or a named constant. Without the quotes, the compiler assumes that the name references a constant and will probably flag a field not defined error.

Therefore, for this fixed form code:

```
D myNewExtDs      E DS           ExtName(PRODUCTX)
D myDataArea       S           20a   DtaAra(JONSDTAARA)
```

You might expect that the conversion would be like this:

```
Dcl-Ds myNewExtDs ExtName(PRODUCT) End-ds;
Dcl-S myDataArea Char(20) DtaAra(JONSDTAARA);
```

But it is not quite that simple and apostrophes must be added around the literals like the following:

```
Dcl-Ds myNewExtDs ExtName('PRODUCT') End-ds;
Dcl-S myDataArea Char(20) DtaAra('JONSDTAARA');
```

Another potential pitfall that you might encounter involves the use of ellipses in data definitions. If you use long descriptive field names in your code you often might have encountered situations where you have needed to use these. For example:

```
D masterAccountDetail...
D                   DS           LikeDS(accountDetail_T)
```

In converting this manually the most obvious approach would be to replace the initial D with Dcl-Ds, remove the D and DS from the second line and you would be done. The resulting code would look like the following, however this won't compile:

```
Dcl-Ds masterAccountDetail...
                  LikeDS(accountDetail_T);
```

The reason is quite simple, the ellipses tell RPG to continue the specification at the first non-blank character of the next line. Therefore, the compiler sees the LikeDS simply as a continuation of the name and a number of errors will result. Ellipses can still be used in free-form, but only to continue the name, not the specification as a whole. In other words, you will only ever need them if you are writing an essay rather than a field or procedure name.

This is what our converted example should look like:

```
Dcl-Ds masterAccountDetail
                  LikeDS(accountDetail_T);
```

Or for that matter it can now simply be a single line:

```
Dcl-Ds masterAccountDetail LikeDS(accountDetail_T);
```

2.2.7 Defining prototypes and subprocedures

Just as with files and data, declarations are also used for defining prototypes, procedures, and procedure interfaces. As with the data declarations, a number of sensible defaults and other enhancements were also implemented. This section discusses some of these declarations using examples.

The different declaration types, together with their associated END-xx definitions, are listed in Table 2-5.

Table 2-5 END-xx declarations

Data Declarations	Definition
DCL-PR	Defines the start of a prototype definition.
DCL-PL	Defines the start of a procedure interface.
DCL-PROC	Begins a procedure definition (that is it replaces the old B(egin) P-spec).
END-PR	Ends the prototype definition. The prototype name can optionally be specified.
END-PL	Ends the procedure interface. The procedure name can optionally be specified.
END-PROC	Ends the Procedure definition (that is it replaces the old E(nd) P-spec). Procedure name can optionally be specified

In some respects being able to code subprocedures completely in free-form is one of the nicest parts of the new support. Having to drop out of free-form into fixed form P and D specs to start a subprocedure, and then going back to fixed form to end it was simply annoying and ugly to boot.

Based on the number of queries that have popped up on Internet lists as to how to code subprocedures in free-form, a lot of people have trouble in this area. We're not sure why because the "rules" are basically the same as for all the free-form declarations.

Start with the DCL-xx, follow it with the name (or *N if you are not naming it), then the item definition (the length and data type that were previously in fixed columns), and finally any keywords from the options area. It is simple if you just remember those basic rules. Following are a few examples to help cement the pattern in your mind.

Prototypes

The basic building blocks for prototypes are DCL-PR, END-PR, and DCL-PARM. As with DCL-SUBF in data structures, the DCL-PARM keyword is only needed when someone has given a parameter the same name as an RPG op-code.

Program prototypes

This is an example of a fixed form version of a program prototype for QCMDEXC:

```
d qcmdexc      Pr          ExtPgm('QCMDEXC')
d   cmd        500a      Const
d                   Options(*VarSize)
d cmdLength     15p 5 Const
d processICG    3a      Options(*NoPass)
```

Here is the free-form counterpart:

```
dcl-pr qcmandc ExtPgm('QCMDDEXC');
    cmd      char(500)  Const Options(*VarSize);
    cmdLength packed(15: 5) Const;
    processICG char(3)      Options(*NoPass)
end-pr;
```

The only real difference between the two examples, apart from the free-form, is the addition of the END-PR directive.

Note that the DCL-PR could also have been coded like this:

```
dcl-pr qcmandc ExtPgm;
```

As you can see, if you code the prototype name to match that of the actual program object, then only the EXTPGM keyword is needed. The program name does not have to be specified. The compiler simply converts the prototype name to uppercase and uses that. This is a good idea as there have been situations where programmers have coded the literal for the program name on the EXTPGM keyword in lowercase. They then wonder why the program call fails. It appears that IBM has also enabled this particular feature in fixed form.

As with data structures, an end directive is always required. However, the END-PR can be coded on the same line as the DCL-PR in cases where there are no parameters. For example if program MYPROGRAM takes no parameters, then the prototype could be coded like this:

```
dcl-pr MyProgram EXTPGM end-pr;
```

Note: In this case, there cannot be a semi-colon (;) after the EXTPGM keyword, it can only be at the end of the line following the END-PR.

Procedure prototypes

Procedure prototypes follow the same basic pattern as those for programs.

One new feature is the introduction of the new keyword *DCLCASE. Those of you who make a lot of use of functions and APIs with mixed-case names, such as Qp0IRenameUnlink and Qp0IRenameKeep, will appreciate this feature. Previously, when prototyping such a procedure, you had to specify the name as a literal to the EXTPROC keyword because without it the compiler would have upper-cased the prototype name and used that, which would not have worked. Now by simply specifying EXTPROC(*DCLCASE) you are telling the compiler to use the procedure name exactly as you declared it on the DCL-PR.

This example demonstrates how you had to code the EXTPROC keyword previously:

```
dcl-pr MixedCaseProcName  ExtProc('MixedCaseProcName')
end-pr;
```

This example shows how *DCLCASE can be used to avoid re-typing the procedure name as it was in the original version:

```
dcl-pr MixedCaseProcName  ExtProc(*DclCase)  end-pr;
```

Defining subprocedures and procedure interfaces

The old beginning and ending P-specs have also been replaced by DCL-PROC and END-PROC respectively.

The new procedure interface definition DCL-PI follows the same pattern as that of the prototype. It uses DCL-PI begin the definition and END-PI to terminate it. DCL-PARM, as with

the prototypes, is used to define any parameters within the procedure interface whose names clash with RPG op-codes.

In fixed form, the procedure name could be omitted from the procedure interface definition. In free-form the place marker *N must be used if the name of the procedure is to be omitted on the DCL-PI directive. This is illustrated in Example 2-3 on page 25.

The example code includes the definition of the data structure that is returned by the subprocedure. The routine accepts a single date in *MDY format. By specifying the CONST keyword, this ensures that any format of date is accepted and formatted appropriately for processing. It returns a data structure containing the day number, day name, and a fully formatted date string.

Note that in the fixed form version /Free and /End-Free had to be used to switch between fixed and free-format. Admittedly in version 7 these directives could be removed, but the ugliness of switching back and forward would still remain.

Example 2-3 Procedure interface

```
// Fixed form version
d dateInfo_T1      DS          Template
d   dayNumber           1s 0
d   dayName            9a
d   dateString         40a  Varying
...
/end-free

// DateInfo Subprocedure
p DateInfo          B          Export
d                   PI          LikeDS(dateInfo_T)
d   inputDate        D          DatFmt(*MDY) Const

/free
// Calculation logic goes here ...
/end-free

p DateInfo          E

// Free-Form version

dcl-ds dateInfo_T  Template;
dayNumber  zoned(1);
dayName    char(9);
dateString varChar(40);
end-ds;

// DateInfo Subprocedure

dcl-proc DateInfo Export;
dcl-pi   *N      LikeDS(dateInfo_T);
      inputDate   date(*MDY) Const;
end-pi;

// Calculation logic goes here ...
end-proc DateInfo;
```

Other ILE related changes

This section does not relate specifically to the use of subprocedures, but rather to ILE in general. Simply put, if you have coded at least one free-form ILE related control statement such as ACTGRP, BNDDIR, or STGMDL, then you no longer need to remember to specify DFTACTGRP(*NO). This is part of the move to sensible defaults for RPG.

Apart from the obvious advantage of being free-form, there is one other major benefit that accrues from this latest support. File and data declarations can now be intermixed. This allows you to define a file together with any related flags, data structures, constants, and so on. In fact, this is even allowed if you stick with the old fixed format F and D specs.

2.2.8 Miscellaneous topics

This section covers the following miscellaneous topics:

- ▶ “Conditional compiler directives” on page 26
- ▶ “The latest updates” on page 26
- ▶ “The rules” on page 27

Conditional compiler directives

The conditional compiler directives feature has been available for many years now. However, with fixed form code this feature could only be used to selectively include or omit individual data declarations or other code blocks. In other words, you could not use them to selectively control part of a definition.

With new support that is now possible. The following is a very simple example based on the idea that you might want to have a single source, but be able to use it to generate both a UTF-16 (double byte) version of the program as well as a conventional single byte version. Using the new support the related variables can be defined like this:

```
Dcl-s accountName
  /If Defined(UTF16)
    ucs2(40) ccsid(1200)
  /Else
    char(40)
  /EndIf
;
```

If when the program is compiled, condition UTF16 was defined, the accountName field is defined as usc2(40), but if that condition was not defined, the field is defined as char(40).

Note: Currently the definition terminating semi-colon has to be placed outside of the conditioned code as shown in this example. In some ways it would be preferable to be able to add it to both definitions (that is following the ccsid(1200) and char(40) entries) in the knowledge that only one will ever be used. Perhaps the compiler will be updated to allow that in the future.

The latest updates

The most recent change to RPG occurred with the IBM version 7.3 release (which was also made available in version 7.1 and 7.2 releases by PTFs). This change removed the final remnant of RPG's punch card heritage by allowing the RPG code to start in any column and to continue for the full length of the source line. There is now effectively no limit to the length of a source line.

The rules

These are the basic rules for utilizing the new support:

- ▶ All source files (including /Copy and /Include files) are assumed to be in the IBM i version 7.2 style.
- ▶ To utilize the full width free-form support, the first line in the source file must begin with **Free in column 1.
- ▶ A **Free source file cannot contain any fixed form code. If this is a requirement (for example to include O-specs) then it must be done using a /Copy.

Support for this feature has also been added to the SQL pre-compiler.

2.2.9 Moving to free-form

Obviously you can manually change your source code to the fully free-form style, but you will almost certainly find it easier to use a tool to help you do the job. At the time of writing this book, there several options available:

- ▶ Arcad Software's ARCAD Transformer RPG. This tool converts source members to free-form, including declarations. A command interface and RDi plug-in are available. You can obtain this tool through the IBM ordering system or from ARCAD directly.

For more information, see the ARCAD website:

<http://www.arcadsoftware.com/resource-items/arcad-transformer-rpg>

- ▶ Linoma Software RPG Toolbox. This tool is the latest evolution of the first RPG III conversion utility and accommodates free-form formatting and much more. A command interface and RDi plug-in are available.

For more information, see the Linoma website:

<http://www.linomasoftware.com/products/rpgtoolbox>

- ▶ Craig Rutledge's JCRCmds open source tools. This tool was updated to include full Free-form conversion

For more information, see the following website:

<http://www.jcrcmds.com>

2.3 Modern RPG programs and subprocedures

A modern RPG program is one that is written by using free-form syntax and is modular in structure. This modular structure is implemented by writing *subprocedures*.

Subprocedures may be coded internally in a module or they can be external (in a service program or another bound module).

Although multiple subprocedures are coded within a module, the approach to writing a subprocedure should be that the subprocedure is stand-alone, which means that the design of a subprocedure is that it makes use of local variables instead of global variables, and that all required data that is external to the subprocedure is passed as parameters or a return value. Simply put, think of every subprocedure as a *stand-alone program*.

This approach to writing subprocedures means that when you determine that a subprocedure might be useful in other places, it is a simple process to remove it from its current module and place it in a service program.

A subprocedure should be designed to perform one task (such as `calculate_Pay()` or `get_customerData()`). The subprocedure can call other subprocedures to achieve that task. Therefore, subprocedures should be short and to the point. A preferred practice is being able to see all the executable code for a subprocedure in a single window in Rational Developer for i.

Also, when you find yourself defining global variables, stop and ask yourself why you are doing so.

2.4 Naming

Before looking at the components of naming conventions, remember that although RPG is a mixed-case language, it is not a case-sensitive language. In an RPG program, the variable names `customerID`, `CustomerID`, `customerid`, and `customerId` all refer to the same variable, and in a PHP or Java program, they are four different variables. However, you should strive to use the same mixed-case form everywhere you use the name.

The following topics are covered in this section:

- ▶ 2.4.1, “The names” on page 28
- ▶ 2.4.2, “Case” on page 28
- ▶ 2.4.3, “Special characters” on page 29
- ▶ 2.4.4, “Underscore” on page 29
- ▶ 2.4.5, “Named constants” on page 29
- ▶ 2.4.6, “Naming conventions” on page 30

2.4.1 The names

Names should be meaningful. The old restriction of 10-character system names on IBM i made RPG programmers masters of abbreviation. It is a habit that must be broken. Names should be meaningful and should not be restricted by a length (although 4096 is the maximum length allowed). Names should be meaningful and descriptive. Just as a name should not be overly abbreviated it should not be overly verbose. For example, the variable should be `customerID`, not `cusID` or `theIDOfTheCustomer`.

When naming variables, arrays, and data structures, think of the name as a noun: It simply states what the item is, for example, `currentAccountNumber`, `customerID`, or `customerList`.

When naming subroutines, subprocedures, or prototype names, think of the name as a verb combined with a noun. There is an action and an item, for example, `calculate_Pay()`, `get_customerData()`, or `convert_toCelsius();`.

Use consistent abbreviations in your names. For example, if you have several procedures that do the “convert” action, be consistent in how you name the action (“convert”, “cvt”, “conv” etc.)

2.4.2 Case

Names (except for named constants) should be mixed case. The usual standard is to use ‘*Camel Case*’. Camel case means that a name is made up of compound words where each word begins with a capital letter. The first word may start with a capital letter or with a lowercase letter, but all following words start with a capital letter. Some examples are `CurrentAccountNumber` or `currentAccountNumber`.

However, because RPG is not case-sensitive, it is up to the programmer to follow the guideline.

2.4.3 Special characters

Except for the underscore character, special characters (such as #, \$, £, and @) should not be used in names. Some special characters change depending on CCSID definitions, so they should be avoided.

2.4.4 Underscore

The underscore character can be used to add clarity to a name. There is an inclination to use underscore to separate compound words in a name, but this is superfluous when CamelCase is used. The name currentAccountNumber is as legible as current_Account_Number.

An underscore can be useful when used with subroutine, subprocedure, or prototype names. The underscore is used to separate the action from the item, such as `calculate_Pay()` or `get_customerData()`.

Look at the following line of code:

```
salary = calculatePay(97);
```

A programmer must check whether salary was being set by a call to the subprocedure `calculatePay()` or from element 97 of the array `calculatePay`. The use of underscore in subprocedure names adds clarity to the code:

```
salary = calculate_Pay(97);
```

Underscore might also be useful when used with named constants and naming conventions.

2.4.5 Named constants

Use named constants instead of literals. Named constants make code self-documenting and easier to maintain. The exception to this is the use of 0 and 1 in expressions when clearing, incrementing, and decrementing field values. A constant name should reflect the function of that constant, not the value.

The convention in most programming languages is that named constants are all uppercase. Accordingly, underscore should be used to separate compound words within the name.

Look at the following literal:

```
if (%status(myFile) = 1218);
```

Now, compare it with the use of a named constant:

```
if (%status(myFile) = ERR_RECORD_LOCKED);
```

If literals are standard throughout an application (for example, status codes), the named constants should be defined in a copy member and included in programs as required.

2.4.6 Naming conventions

With the provision that names should be meaningful, naming conventions can be used to identify a usage or grouping of variables, or named constants. The convention is that the correlated names start with the same characters followed by an underscore.

For example, a copy member includes the following named constants that are used to identify message IDs. All of the named constants begin with MSGID_.

```
dcl-C MSGID_CODE      'ERR0001';
dcl-C MSGID_DESCRIPTION 'ERR0002';
```

Use a standard prefix or suffix to distinguish template variables and files. For example, use a prefix such as type_, or typ_, or t_, or use a suffix such as _T, or _typ, or _type.

For the rare occasion where global variables are used in subprocedures, the names of the global variables begin with the characters g_.

When naming subprocedures and prototyped program calls, it is imperative that the naming convention be consistent. For example, subprocedures that add information to a database should start with add_ or write_, not a mixture of the two.

Naming conventions are definitely required for names that are defined through copy members or globally defined in a program/module. But the use of local variables (in subprocedures) and qualified data structures minimizes the requirement for naming conventions within subprocedures.

2.5 Comments

All programs must be documented. One of the major benefits of free-form RPG and correct naming conventions is that it reduces the need for detailed documentation because the code is self-explanatory. Even so, programs must be documented.

Comments should be used in two ways in RPG programs and subprocedures:

- ▶ Summary for a program, subprocedure, or section of code
- ▶ Detailed commenting of certain code

2.5.1 Summary comments

Summary comments should be at the start of every program and subprocedure. Summary comments should contain, at least, the following information:

- ▶ The title of the program/subprocedure
- ▶ A description of what the program/subprocedure does
- ▶ The name of the author
- ▶ History of changes that are made to the program

2.5.2 Detailed commenting

One of the major benefits of free-form RPG and correct naming conventions is that it reduces the requirement for detailed commenting. Detailed commenting should be required only to explain some complex coding technique or to highlight a technique being used in the code.

2.5.3 Other commenting

Use blank lines to group and segment code.

It can be useful to use a marker line comment to separate major sections of a program, although the requirement for this is nullified when you use the Filter View feature in Rational Developer for i, as shown in Figure 2-1.

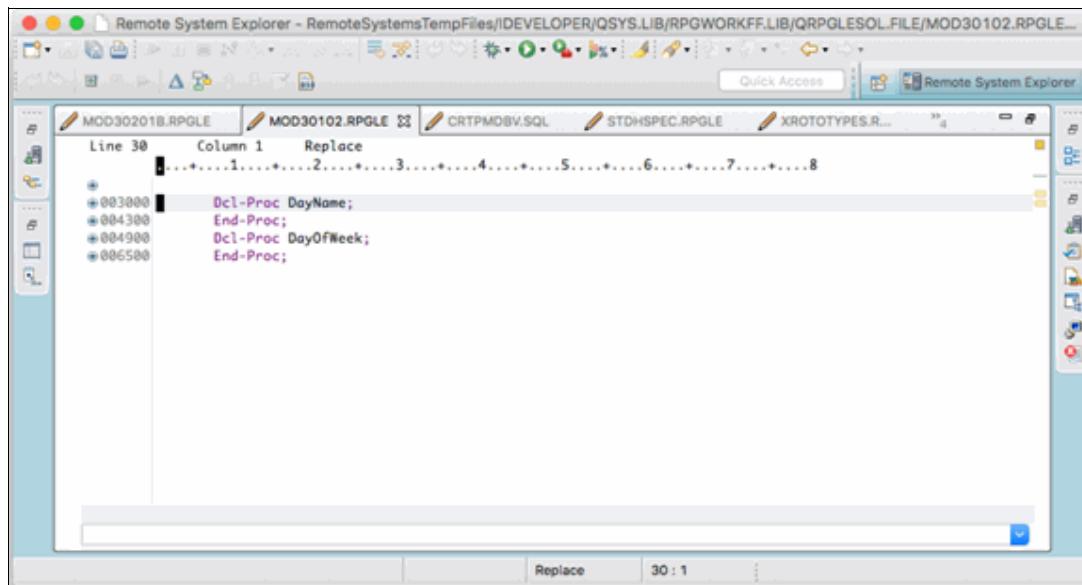


Figure 2-1 Filter view feature in Rational Developer for i

2.5.4 Positions 1 - 5

Historically, positions 1 - 5 were used to indicate or flag lines that were changed for a certain modification. This practice should be avoided. Specifying **FREE on the first line of code means that positions 1 to 5 on all subsequent lines may now be used for code.

2.6 Structuring code

All code should be structured. The standards and guidelines for declarative code and executable code are slightly different.

2.6.1 Declarative code

Declarative code is defined at the start of a module, program, or subprocedure. Definitions should be grouped together by type of declaration.

Declarations should be grouped so that related items are defined together. The procedure interface should be first, before any other declarations.

Indentation should be used with data structured to identify overlaying structures.

Align definitions so they are easy to read. For example, when defining stand-alone variables, parameters or data structure subfields, align the data type on each line.

Compare the definition of this data structure with alignment, as shown in the following example code and then without alignment:

```
dcl-Ds APIError qualified;
  bytesprovided int(10) inz(%size(APIError));
  bytesavail   int(10) inz(0);
  msgid        char(7);
  *N           char(1);
  msgdata      char(240);
end-Ds;
```

To one without alignment

```
dcl-Ds APIError qualified;
bytesprovided int(10) inz(%size(APIError));
bytesavail   int(10) inz(0);
msgid        char(7);
*N           char(1);
msgdata      char(240);
end-Ds;
```

2.6.2 Executable code

All executable code should be indented. Indentation within loops and groups add to the legibility of the code. Here is an example:

```
if (messageCount() = 0);
  select;
    when CGIOption = 'CANCEL';

    when CGIOption = 'DELETE';
      failed = delete_Event(persistId);
    other;
      setEventData(persistId : data);
      if (messageCount() = 0);
        failed = put_Event(persistId);
      endif;
    endS1;
  endIf;
```

If a statement takes more than one line of code, use alignment to make the code more legible, as shown in the following code:

```
set_days_for_Event(data.event:
  %date(data.fromdate: *USA):
  %date(data.todate: *USA):
  %date(data.wrkshpdate: *USA));
```

Automatic indent (and closure of control block) can be set in the ILE RPG preferences for the Remote Systems LPEX Editor in Rational Developer for i, as shown in Figure 2-2.

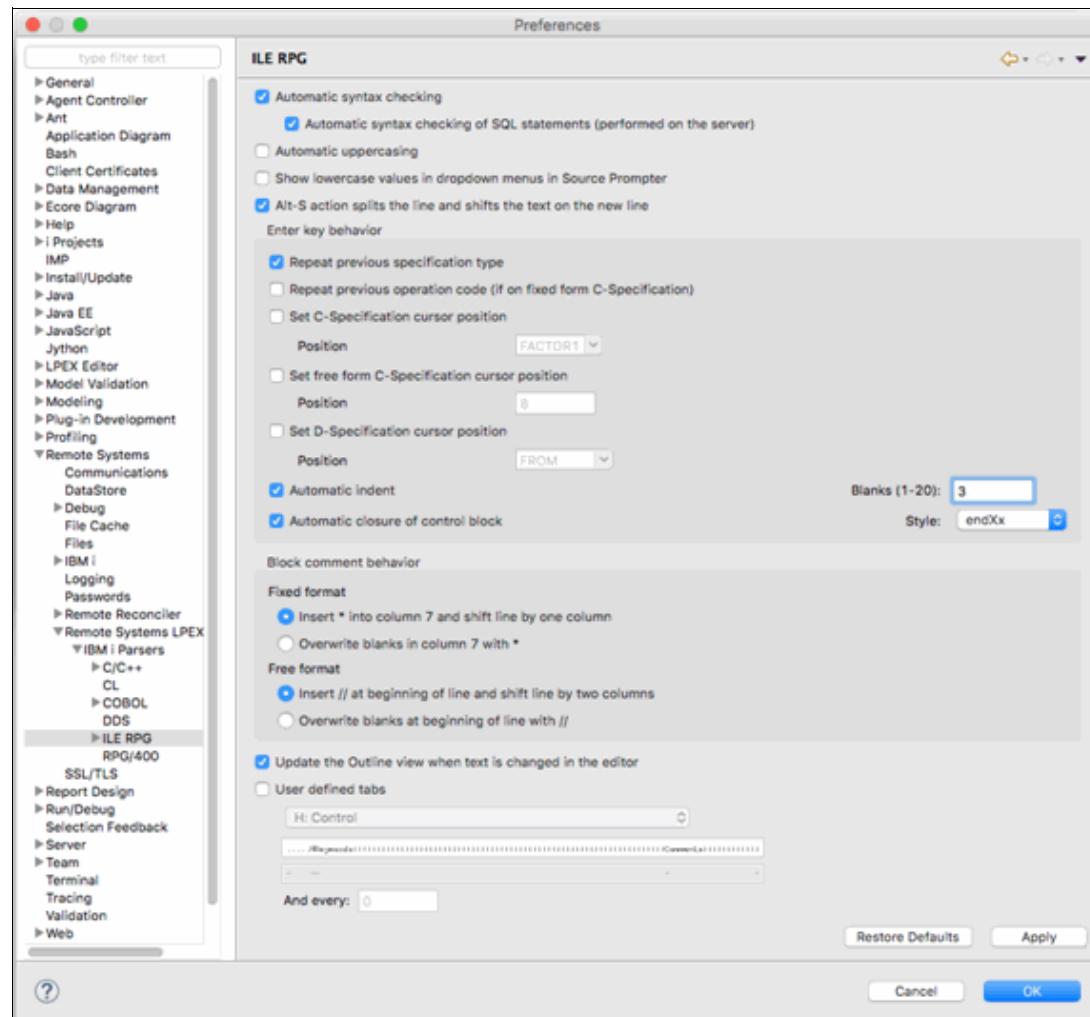


Figure 2-2 Setting automatic indentation in Rational Developer for i

2.6.3 Multipath comparison

Deeply nested IF/ELSE/ENDIF code blocks are hard to read and result in an unwieldy accumulation of ENDIFs at the end of the group. Instead use the more versatile SELECT/WHEN/OTHER/ENDSL or the IF/ELSEIF/ENDIF constructions.

Use SELECT/WHEN if the choice statements all compare a particular variable to a set of values. Use IF/ELSEIF if the choice statements have varied types of conditions.

2.6.4 Embedded SQL

Use indentation and alignment to make SQL statement legible. An SQL formatter is available in Run SQL Scripts in IBM i Access Clients Solutions that is included in Rational Developer for i. The following is an example of well formatted SQL:

```
exec sql
  declare C001 scroll cursor for
    select event, daynum, agendaid, fromtime, totime, showseq, title
    from AGENDA
    where event = :eventIn
    order by event, daynum, showseq, agendaid
    for read only;
```

2.7 Using templates and qualified data structures

Templates and qualified data structures provide a means of clearly defining and documenting data items in a program.

A qualified data structure means that all references to the data structure subfields must be qualified with the data structure name (for example, `mailAddress.city` and `mailAddress.state`) so that subfields with the same name can be defined in multiple data structures without the possibility of conflict and so that the association between a subfield and its data structure is clear.

Defining a qualified data structure as a template means that the data structure may not be used as a data structure, but can be used as the template for other data structures that are defined by using the `LIKEDS` keyword. In the following example, a reference to the subfield `baseAddress.city` is invalid, but a reference to the subfield `mailAddress.city` is valid:

```
dcl-Ds baseAddress template qualified;
  street1 char(30);
  street2 char(30);
  city   varchar(20);
  state  char(2) inz('MN');
  zip    char(5);
  zipplus char(4);
end-Ds;

dcl-Ds mailAddress likeds(baseAddress) inz(*likeDS);
```

Templates and qualified data structures provide an excellent means of gathering together related “work” variables in a program or providing parameters for a call interface. The definition of the template data structure can be placed in the same copy member as the prototype definition for a called program or subprocedure.

You can use the `LIKEREc` keyword to define qualified data structures based on input/output records for an external described file.

Additionally, you can also define template fields (with the `LIKE` keyword) and files (with the `LIKEFILE` keyword)

2.8 Qualifying wherever possible

When using built-in functions such `aEOF()`, `%FOUND()`, `%EQUAL()`, and `%STATUS()`, always provide the associated file name as a parameter.

Unfortunately, the `%ERROR()` built-in function does not allow for a file name parameter, so always check the `%ERROR()` function (or the `%STATUS()` function) directly after an operation with an error (E) extender.

2.9 Strings

When it comes to string handling, it is better to use varying length (VARCHAR) as opposed to character (CHAR) fields. The use of varying length fields reduces the requirement for string functions (such as `%TRIM`) and makes the code more legible. As you do more programming targeted for the Web, strings become more likely to be leverage.

2.10 Subroutines

Subroutines should not be used for modularization/structure - subprocedures should be used instead. But subroutines can be useful in organizing the logic in a subprocedure.

2.11 Older functions

When using only free-form RPG, many of the old RPG functions are no longer available, for example, operation codes such as `MOVE`, `MOVE1`, `GOTO`, `ADD`, and `SUB`, conditioning, and resulting indicators. But, there are still some old features that are available but should be avoided.

2.11.1 RPG built-in indicators

Using RPG numeric indicators (*IN01 - *IN99) and special indicators (*INU1 - *INU8) should be avoided at all costs. They are not self-explanatory and dependent on comments to clarify their usage.

Define your own indicators when a Boolean condition is required. For example, use:

```
if monthEnd;
```

instead of:

```
if *in70;.
```

If the program externally described display files or print files that use numeric conditioning or resulting indicators, then define an Indicator Data Structure for the display/print file, which remaps the numeric indicators (in the display/print file) to indicators with meaningful names in the program.

Here is example code to accomplish this task:

```
dcl-F Mod30101D workstn(*ext) usage(*input:*output) IndDs(WSI);
dcl-Ds WSI qualified;
    F3Exit      ind pos(3);
    F5Refresh   ind pos(5);
    F12Cancel   ind pos(12);
    F23Delete   ind pos(23);
    pageDown    ind pos(26);
    pageUp     ind pos(27);

    errorInds   char(10) pos(31);
    enableDelete ind pos(41);

    SFLInds    char(3) pos(51);
    SFLDsp     ind pos(51);
    SFLDspCtl  ind pos(52);
    SFLClr     ind pos(53);

    SFLNxtChg  ind pos(54);
    SFLPageDown ind pos(55);
    SFLPageUp   ind pos(56);
    SFLProtect  ind pos(57);

    enableMsgSFL ind pos(91) inz(*on);
end-Ds;
```

2.11.2 Compile-time arrays

The definition of a compile-time array means that the definition of the array (in the data declarations) is separate from the definition of the data (which is at the end of the program), as shown in the following example:

```
dcl-S monthNames char(9) dim(12) ctData perRcd(3);
// (lots and lots and lots of code here)
    ...
**CTDATA MonthNames
January  February March
April    May       June
July     August    September
October  November December
```

It is better to define an array and its corresponding data in a data structure, with the definition of the data structure overlaying the definition of the data, as shown in the following example:

```
dcl-Ds allDays;
*N      char(9) inz('January');
*N      char(9) inz('February');
*N      char(9) inz('March');
*N      char(9) inz('April');
*N      char(9) inz('May');
*N      char(9) inz('June');
*N      char(9) inz('July');
*N      char(9) inz('August');
*N      char(9) inz('September');
*N      char(9) inz('October');
```

```

*N      char(9)  inz('November');
*N      char(9)  inz('December');
monthNames char(9) dim(12) pos(1);
end-Ds;

```

The definition of an array in the same location as the data makes it easier to modularize code into subprocedures, when required.

2.11.3 Multiple occurrence data structures

Use data structure arrays in place of multiple occurrence data structures. Multiple occurrence data structures allow access only to one occurrence (element) at a time (it cannot directly compare two occurrences of a multiple occurrence data structure), and using them to set the required occurrence (**OCCUR/%OCCURS()**) is cumbersome, as opposed to using an index to identify an array element.

2.11.4 Do not use GOTO, CABxx, or COMP

Instead substitute a structured alternative, such as nested IF statements, or status variables to skip code or to direct a program to a specific location. To compare two values, use the structured opcodes IF, ELSE, and ENDIF. To perform loops, use DO, DOU, and DOW with ENDDO. Never code your loops by comparing and branching with COMP (or even IF) and GOTO. Employ ITER to repeat a loop iteration. and use LEAVE for premature exit from a loop.

2.11.5 Do not use obsolete IFxx, DOUxx, DOWxx, or WHxx opcodes

The newer forms of these opcodes - IF, DOU, DOW, and WHEN - support free-format expressions, making these new alternatives more readable. In general, if an opcode offers a free-format alternative, use it.

2.11.6 Use SELECT, WHEN, OTHER, ENDSL for multipath comparison

Deeply nested IF/ELSE/ENDIF code blocks are hard to read and result in an unwieldy accumulation of ENDIFs at the end of the group. Instead use the more versatile SELECT/WHEN/OTHER/ENDSL construction.

2.12 Embedded SQL

Apart from the standard style guidelines, there are a few guidelines that relate specifically to embedded SQL:

- ▶ Keep SQL statements as simple as possible. You do not want to debug a complex SQL statement in an RPG program. Create a view that “hides” the complexity of joins and casting and select from that view in the RPG program.
- ▶ Avoid naming variables that start with SQL. They might conflict with variable names that are automatically included by the SQL pre-compiler.
- ▶ Use **SET OPTIONS** to ensure that the SQL environment is specified correctly at compile time.
- ▶ Wherever possible, use multi-row fetch instead of single-row fetch.

2.13 Global definitions

Global definitions should be kept to a minimum.

The most valid candidates for global definitions are file and data area definitions. If they are used, they should be modularized within a module with the subprocedures that are processing them. Consider using qualified data structures for all I/O.

If global variables are being defined, they should be qualified (either in a data structure or with a prefix such as gv_), so they are easily identified as global variables within a subprocedure.

The use of the **IMPORT** and **EXPORT** keywords for variables should be used only as a last resort and be documented when used.

2.14 Parameters and prototyping and procedure interfaces

As well as providing a means of validating parameter definitions at compile time, you can use prototyping and procedure interfaces to specify how parameters are used.

Prototypes are required only for external subprocedure or program calls; they are not required for subprocedures that are coded and called only within the same module/program.

2.14.1 Parameters

If the value of a parameter is not changed by a subprocedure/program, use the **VALUE** or **CONST** keywords to indicate that such is the case. These stop parameters from being inadvertently changed by a subprocedure or program

2.14.2 Return value

One of the items that is often debated is whether subprocedures should always return a value.

Subprocedures can be used to code procedures (that do not return a value) and functions (that do return a value).

A call to a program is a call to a procedure because programs cannot return a value.

Conceptually, a procedure is code that takes (optional) parameters and performs an action, and a function is code that takes (optional) parameters and “calculates” a return value.

The argument for always returning a value is that there is always a value to return. For example, even if a procedure does not calculate and return a specific value, should it return a value to indicate whether the process worked?

As a preferred practice, differentiate between procedures (which do not return a value) and functions (which do return a value).

2.14.3 Copy members

All prototypes should be coded in copy members and include required programs and modules. A prototype should never be coded in more than one source member.

The copy members that contain the definition of the prototypes should also contain the definition of any templates or named constants that refer specifically to the use of the corresponding prototypes.

Managing the maintenance and usage of the prototype members can be quite a challenge and (at the moment) there is no single simple solution. Every solution comes with a cost.

The prototype members must be easy to maintain - you do not want a situation where two programmers need to change prototype definitions in the same member at the same time. This means that there will usually be a one to one correspondence between prototype members and modules - i.e. a prototype member containing all prototypes, templates and named constants for subprocedures in a corresponding module. The same one to one correspondence would apply to prototypes for program calls although a "program" prototype member might contain prototypes for a group of related programs.

The ease of maintenance requirement means there will be quite a few prototype members. There is now the challenge of how the programmer knows which members to include in a program when a call is to be made to a program or an external subprocedure. This is even more challenging if the prototype members are in a source physical file with a ten character naming restriction - there is little chance of having meaningful names. An alternative is to define prototype members in directories in the Integrated File System (IFS), where meaningful names can be given to the prototype members.

Compare the following directive to include a prototype member from a source file:

```
/include qrpglesrc,UTILITY01P
```

with the corresponding directive for a file in the IFS:

```
/include '/myApp/proto_utility/userSpaceAPIs.rpgle'
```

The requirement for /INCLUDE directives can be reduced by using nested copies. It is possible to have a single /INCLUDE directive that would include all prototype members in the application. Whereas this approach is appealing (the programmer only needs to know the name of one copy member), there are a number of considerations:-

- ▶ Someone must be responsible for maintaining the extra copy members which handle the nesting and grouping of the prototype members.
- ▶ All prototypes, templates and named constants (defined in prototype members) will now be included in the Outline View in Rational Developer for i. If there are a lot of prototypes, templates and named constants - the Outline View will take a long time to refresh. This can be somewhat alleviated through the use of conditional compilation directives (as you will see in a following example) but that, in turn, leads to even more complicated maintenance of the extra copy members which handle the nesting and grouping of the prototype members.
- ▶ Change management systems may have difficulty when a change is made to prototype member (even as simple a change as adding a comment). The change management system may determine that, since the member is included in every program and the member has changed then every program should be re-created.

The following example shows one approach to using nested copies to minimize the requirement for multiple /INCLUDE directives:

```
/include common,baseInfo
```

The BASEINFO member contains nested include directives, as follows:

```
/include utility,pUtility
/include fileProcs,protoFile
/include common,commproto
/include genstat,pStatGen
/include regFunc,pRegFunc
//-----
// Include CGI Prototypes, if required
/If Defined(CGIPGM)
/include CGIDEV2/QRPGLESRC,PrototypeB
/include CGIDEV2/QRPGLESRC,UseC
/EndIf
//-----
// Include HSSF Prototypes, if required
/If Defined(HSSF)
  include SIDSrc,HSSF_H
/EndIf
```

Each of the included members might contain prototype definitions or further nested include directives. For example, the PUTILITY member contains nested include directives as follows:

```
/include utility,PUTILMSG
/include utility,PUTILCGI
/include utility,PUTILSPACE
/include utility,PUTILIFS
/include utility,PUTILDATE
```

The members PUTILMSG, PUTILCGI, PUTILSPACE, PUTILIFS, and PUTILDATE contain the prototypes.

One of the difficulties with this approach is that many prototypes (and templates) are being included. Although this has no effect on the size of the program/module when it is compiled (prototypes and templates are declarative), it can take a long time for the outline view in Rational Developer for i to refresh.

This situation can be solved by using compiler directives to indicate what should and should not be included. Changing the definition of the PUTILITY member as follows means that a definition name must be set in order for the prototypes to be included:

```
/if defined(UTILITY)
#define UTIL_MESSAGE
#define UTIL_CGI
#define UTIL_USERSPACE
#define UTIL_IFS
#define UTIL_DATES
/endIf
/if defined(UTIL_MESSAGE)
/include utility,PUTILMSG
/endIf
/if defined(UTIL_CGI)
/include utility,PUTILCGI
/endIf
/if defined(UTIL_USERSPACE)
/include utility,PUTILSPACE
/endIf
/if defined(UTIL_IFS)
/include utility,PUTILIFS
```

```
/endIf
/if defined(UTIL_DATES)
/include utility,PUTILDATE
/endIf
```

The originating program, which includes the BASEINFO member, can set the required definition names or can define UTILITY, which results in all members being included:

```
/define UTIL_MESSAGE
#define UTIL_USERSPACE
/include common,baseInfo
```

The difficulty with this approach is that the programmer must know the required definition names, so documentation is required.

The benefit of this approach is that it documents what is being included in the program.

Note that in this example, definition names are being used to include single members, but they could just as easily be used to include multiple members (as with CGIPGM in the BASEINFO member).

As stated earlier, there is no simple solution to managing the maintenance and usage of the prototype members. The challenge is to find which combination of techniques will provide the best balance between ease of maintenance and ease of use

2.15 The integrated language environment

There are many approaches to setting standards for the integrated language environment or ILE environment. Here are some common questions:

- ▶ How many service programs should you have?
- ▶ How many modules should there be in a service program?
- ▶ How many modules should there be in a program?
- ▶ When should you use bind by copy and bind by reference?
- ▶ How many binding directories should you have?
- ▶ How should you control signature violations?
- ▶ How many activation groups should you have?

Unfortunately, the answer to all of these questions is: It depends. The structure of an ILE application depends on the application and what it does.

One example of how to structure an ILE application can be found in *Development Environments*, found at:

<http://www.itjungle.com/fhg/fhg051210-story01.html>

A change management system is something that can have a major impact on the methodology that you use when developing an ILE application because the change management system might have a preferred technique for managing service programs, binder language, and so on.

Regardless of the final development environment, the following sections describe guidelines for an ILE environment.

2.15.1 ILE programs

Although the structure of ILE programs can be more complicated (one or more modules or binding to service programs), you still want the process of creating a program to be a simple one. To compile a program, the programmer should not need to know about activation groups and what all the service programs are, much less what subprocedures are in what service programs.

This task can be achieved through the diligent use of binding directories and a standard control specification, which is included in every program through an **/INCLUDE** directive:

```
Ctl-Opt debug datEdit(*MDY/) option(*srcStmt:*noDebugIO) bndDir('MYAPP');
/if defined(*CRTBNDRPG)
Ctl-Opt dftActGrp(*no) actGrp('PGMBND');
/endIf
```

2.15.2 Service programs

Service programs are at the core of any ILE application. If a subprocedure can be used in more than one place, then it belongs in a service program.

Because the content of a service program can be called from multiple places, the management of a service program requires a bit more care than “normal” programs. The approach to managing changes to a service program should be along the same lines as the way changes to a database are managed. As with a database, the minimum of people should have the ability to make changes. Any programmer can develop a subprocedure, but not any programmer can incorporate it in a service program.

Here are some basic pointers for service programs:-

- ▶ How many service programs should an application have? There is no magic number. Each service program should contain groups of related functions: utilities, database processing, API handlers, and so on. For ease of maintenance, some service programs might be split into multiple service programs (for example, database processing).
- ▶ As with service programs themselves, the number of modules per service program and the number of subprocedures per module should be determined by related functions and ease of use. Ease of maintenance is the key.
- ▶ Keep the source members for a service program separate from other source members. Maybe use a source file name that has the same name as the program.
- ▶ Determine how you will manage prototype/template members for the contents of the service program (one copy member, a copy member per module, and so on).
- ▶ Consider having a separate binding directory for each service program. The bindings that are required for a service program can be different from the bindings that are required for an application program.
- ▶ Always use binder language. *Never* use **EXPORT(*ALL)** when creating a service program. Binder language means more maintenance (when subprocedures are added/removed), but it also allows the greatest flexibility in managing the interface to the service program.
- ▶ When adding parameters to a subprocedure, add the parameters to the end of the parameter list by using **OPTIONS(*NOPASS)**, which minimizes the requirement of re-creating anything that calls the changes subprocedure.

2.15.3 Binding directories

When creating an ILE program or service program, binding directories provide a means of generically providing a list of objects (modules or service programs) that may be required for binding.

Binding directories should be kept to a minimum.

A single binding directory should list all service programs (and modules) that might be required when a standard program is created.

Depending on the complexity and cross referencing between service programs, each service program might also require a binding directory.

If there are multiple ILE applications and there are service programs and modules that are common to all of them, then another binding directory may be used to list those objects.

A list of binding directories can be included in the **BNDDIR** keyword on a Control Spec.

There are cases where an application has a binding directory per program. The binding directory lists the modules and service programs that are required to define the program. This is a mistake and imposes a maintenance impact that is not required.

Managing service programs with many modules has to be done either by having a specific binding directory for the service program, by using something like CL to create the service program or by using a generic name for all modules in the service program.

2.15.4 Activation groups

Generally speaking, an activation group applies to an application. At times, a separate activation group might be used if you are scoping for commitment control or file overrides.

ILE programs should never be run in the default activation group. This situation can be achieved only if an ILE program is created with an activation group of *CALLER and the program is called by an OPM program (or from a command line). It is preferable to use the name of the activation group when creating programs, which is easily achieved by using the **ACTGRP** keyword in a standard Control Spec in the programs.

Even worse than having an ILE program in the default activation group is having a service program in the default activation group. This situation can happen only if an ILE program is running in the default activation group. Again, ILE programs should never be run in the default activation group.

Also, avoid using the QILE activation group. This is the activation group that is used when care has not been taken to choose the activation group. If care was not taken to choose the activation group, care may not have been taken in other aspects of the application, and your application may be subject to the activation group ending unexpectedly, or to overrides and commitment control actions that you do not want.



Subprocedures

One of the major enhancements of RPG IV in V3R2 and V3R6 was the ability to define multiple procedures per module. This chapter discusses how to use those procedures efficiently and also gives a working example on moving from subroutines to subprocedures easily.

The following topics are discussed in this chapter:

- ▶ 3.1, “Subprocedure terminology” on page 46
- ▶ 3.2, “Advantages of using subprocedures” on page 47
- ▶ 3.3, “The anatomy of a subprocedure” on page 48
- ▶ 3.4, “Moving from subroutines to subprocedures” on page 53
- ▶ 3.5, “Using subprocedures efficiently” on page 59
- ▶ 3.6, “More on subprocedures” on page 66

3.1 Subprocedure terminology

This section differentiates multiple terms often used when discussing subprocedures. The different terms explained are:

- ▶ Integrated Language Environment (ILE) modules
- ▶ Main procedure
- ▶ Built-in functions
- ▶ Subroutines

3.1.1 ILE modules

Processing within ILE programs (*PGM) occurs at the procedure level. All ILE programs consist of one or more modules (*MODULE), which in turn, consist of one or more procedures.

You can think of RPG IV modules as falling into one of three basic categories:

- ▶ Those containing only a main procedure and no subprocedures.
- ▶ Those containing a main procedure and one or more subprocedures.
- ▶ Those with no main procedure but with one or more subprocedures.

3.1.2 Main procedure

A main procedure can be called either by a dynamic call (CALL) or by a bound call (CALLB).

Subprocedures, on the other hand, must always be called by a bound call. It is for this reason that subprocedures can only be used in “real” ILE programs. This refers to programs that are created by the Create Program (**CRTPGM**) or by the Create Bound RPG Program (**CRTBNDRPG**) commands.

Subprocedures differ from main procedures in several respects. The main difference is that subprocedures do not (and cannot) use the RPG cycle while running, even if they are part of a module where the main source section is using the RPG cycle.

For a full description of other differences, see the *Programming IBM Rational Development Studio for i, ILE RPG Reference* manual:

https://www.ibm.com/support/knowledgecenter/api/content/nl/en-us/ssw_ibm_i_72/rzasd/sc092508.pdf

3.1.3 Built-in functions

Subprocedures can be defined to return a value, in which case they are often referred to as *functions* or *user-defined functions*. In this case, they are invoked by being referenced in a calculation spec just as if they were an RPG IV Built-in Function (BIF).

3.1.4 Subroutines

Subroutines are different from subprocedures, but they are often referred to as using the same base functionality in the basics of structured programming.

A subroutine is part of the main procedure or any subprocedure (like the *PSSR subroutine) and can be invoked multiple times from different locations (only in the same procedure).

Subroutines share global variables or local variables within the same procedure. The subprocedure offers the same base functionality as subroutines, plus many more advantages, which are discussed in the following section 3.2, “Advantages of using subprocedures”.

3.2 Advantages of using subprocedures

Subprocedures offer a number of significant advantages over subroutines:

- ▶ Subprocedures can define their own local variables.

Local variables can only be modified by logic within the subprocedure. Contrast this with variables that you might have defined for use within a subroutine. Such variables are accessible from anywhere in the entire source member and are, therefore, quite likely to be modified by accident.

- ▶ Subprocedures can accept parameters.

If you want to use parameters in a subroutine, you have to fake them out by defining variables to serve the purpose. This tends to make the code less intuitive since there is no obvious connection between the parameter and the subsequent exit subroutine (EXSR) instruction. Of course, you can always rely on the accurate comments in your code to clear up any possible misunderstandings.

Parameters to a subprocedure can also be passed by value. That is, a copy of the parameter's content (its value) is passed to the subprocedure. This is in contrast to the normal method of parameter passing, where parameters are passed by reference. This means that a pointer to the data is passed rather than the actual data itself.

Variable length parameters can also be used when calling subprocedures.

- ▶ Subprocedures provide parameter checking.

Creating a prototype for the subprocedure allows the parameters passed to that subprocedure to be checked at compilation time for consistency. This helps to reduce run-time errors, which can be costly.

Parameters can be defined as optional or omitted.

- ▶ Subprocedures can be used as a user defined function.

Subprocedures that return a value can be used anywhere in the calculation specifications (specs) that a variable of the same type and size can be used. This allows for an English like interface to the subprocedure that is far more intuitive than an EXSR.

Spec: Throughout this book, the word “spec” refers to *specification*. This applies to all occurrences of this word.

- ▶ You can call the subprocedure from outside the module, if it is exported.
- ▶ Subprocedures provide multiple entry points within the same RPG module.

Multiple exported subprocedures can be placed into the same module and each subprocedure can be called from outside or within the same module. This support is provided in conjunction with the ILE service program functionality.

- ▶ Subprocedures support recursion.

Variables in a subprocedure are by default automatic. A new version of the variable is created each time the subprocedure is invoked. Because of this, RPG IV subprocedures are allowed to recurse, meaning to call themselves directly or indirectly. This can be useful in certain types of design, for example, when building an inventory system that contains multiple levels of parts nested to each other (parts explosions).

3.3 The anatomy of a subprocedure

This section introduces you to the basics of coding subprocedures.

3.3.1 Subprocedure definition

Subprocedures are specified in the following situations:

- ▶ After the main source section, if one exists.
- ▶ Following a control spec containing the NOMAIN keyword, if there is no main section.

Note: There might be other specs between the control spec and the beginning of the subprocedure, for example, file specs to define files to be used by the subprocedures.

Here are the basic elements used when coding subprocedures in your programs:

- ▶ Prototype (see “Prototypes” on page 49 for more information)
- ▶ Procedure Begin and End specs
- ▶ Procedure-Interface (PI) definition
- ▶ Definition specifications of local variables (optional)
- ▶ Calculation section, which incorporates the optional return value

These elements are highlighted in the simple subprocedure shown in Example 3-1, which receives two integers passed as parameters and returns the sum of the two.

Example 3-1 Simple subprocedure

```
// PROCEXAM from QRPGLESRC

// Procedure AFunction which receives 2 integer values and
// returns the sum as an integer

dcl-Proc Afunction; (1)

dcl-pi  AFunction  packed(10 : 0); (2)
        AnInputParm1   packed(5 : 0);
        AnInputParm2   packed(5 : 0);
end-pi;

Dcl-s AReturnValue  packed(10 : 0); (3)
AReturnValue = AnInputParm1 + AnInputParm2; (4)
Dsply AReturnValue;
Return AreturnValue; (5)

end-proc; (6)
```

The markers (1) through (6) on the left side of Example 3-1 on page 48 are defined as follows:

1. The procedure spec supplies the name of the subprocedure. Its presence also signals to the compiler not to flag the subsequent data and calc specs as out of sequence.
2. The Procedure Interface definition identifies the data type (packed decimal) and size (10 digits with no decimal places) of the return value. It also marks the beginning of the parameter list. In this sense, it performs a similar function to the *ENTRY PLIST operation.
- In this example, there are two parameters, **AnInputParm1** and **AnInputParm2**. The end of the list is signaled by the appearance of the **end-pi**;
3. Definitions of variables, constants, and prototypes needed by the subprocedure. These definitions are local definitions.
4. Any calculation specs needed to perform the task of the procedure. The calculations can refer to both local and global definitions, although the use of global data is discouraged. If any subroutines were included within the subprocedure, they are local and cannot be used outside of the subprocedure.
5. If the subprocedure returns a value, supplying the actual value is the task of the **RETURN** operation.

Note that **RETURN** can either use a simple variable, as in this example, or can return an expression. Had we chosen to use this option, we could have simplified this example by coding the following line:

```
Return      AnInputParm1 + AnInputParm2;
```

6. The End procedure spec (**end-proc**). Note that the name of the subprocedure can be repeated here, but is not required.

Prototypes

To call a subprocedure, you should use a prototyped call. You can call any program or procedure that is written in any language in this way. A prototyped call is one where the call interface (the number, type, and size of the parameters) is checked at compile time by reference to the prototype.

The prototype for the sample subprocedure would look something like the following:

```
// Procedure example prototype
dcl-pr  AFunction  packed(10 : 0);
        Packed5      packed(5 : 0);
        Packed5      packed(5 : 0);
end-pr;
```

Note: The field names for the two parameters are identical. Normally this is not permitted in RPG IV, but in this case, it is acceptable since the compiler is going to ignore the name anyway. The only part that matters to the compiler is the number of parameters and their data type and size. However, it is considered a best practice that you do not do this!

A prototype is a definition of the call interface. It contains the following information:

- Whether the call is bound (procedure) or dynamic (program).
- How to find the program or procedure (the external name).
- The number and nature of the parameters.
- The parameters that must be passed, and which parameters are optionally used.
- Whether operational descriptors should be passed.
- The data type of the return value (optional, for subprocedures only).

A prototype must be included in the definition specs of any external program or procedure that makes the call. The prototype is used by the compiler to call the program or procedure correctly, and to ensure that the caller passes the correct parameters. For this verification to occur, the prototype also needs to be included when compiling the module in which the subprocedure is located, and when compiling any module that wants to use the subprocedure.

No prototype is required if the subprocedure is used only within the source member it is defined in. However, it is considered a best practice that you prototype it anyway, that way if you decide to export it at a later date the prototype already exists.

Language interoperability: This section concentrates on writing and using RPG IV subprocedures. The same prototype-writing skills can be applied to call procedures written in other languages, most notably C. This means that RPG IV programs now have access to all the functions in the C function library, which is shipped as part of IBM i on all systems. Other system APIs, previously available only to C programmers, such as TCP/IP sockets, the SSL APIs, and direct program access to the Integrated File System (IFS), are also enabled by the prototyping support associated with subprocedures.

3.3.2 Procedure-interface definitions

For all subprocedures, you need to define a procedure interface. A procedure interface definition is basically a repeat of the prototype information within the definition of a procedure. It is used to define the entry parameters and the return value for the subprocedure. The compiler uses this information to ensure that the internal definition of the procedure is consistent with the external definition (the prototype).

The procedure-interface definition can be specified anywhere in the definition specs of a procedure. In practice, most programmers tend to place them at the beginning of the definition specs. A procedure interface is compulsory if the procedure returns a value, or if it has any parameters. Otherwise, it is optional.

A procedure interface can also be used for the main procedure, in place of the *ENTRY PLIST. This can be coded anywhere in the definition specs after the procedure statement. The procedure spec should be placed after the prototype definitions and any global variables that are going to be used in the program.

3.3.3 Order of coding the source elements

As noted earlier in this chapter, an RPG IV module consists of an (optional) main procedure and zero or more subprocedures. A main procedure is one that can be specified as the program entry procedure and receive control when the program is first called. The main procedure consists of the set of control, file, data, input, calculation and output specs that begin the source.

Any files and global variables that are required in the main procedure or in the subprocedures must be defined in the main section of the code.

Figure 3-1 illustrates the layout of a complete RPG source containing multiple subprocedures.

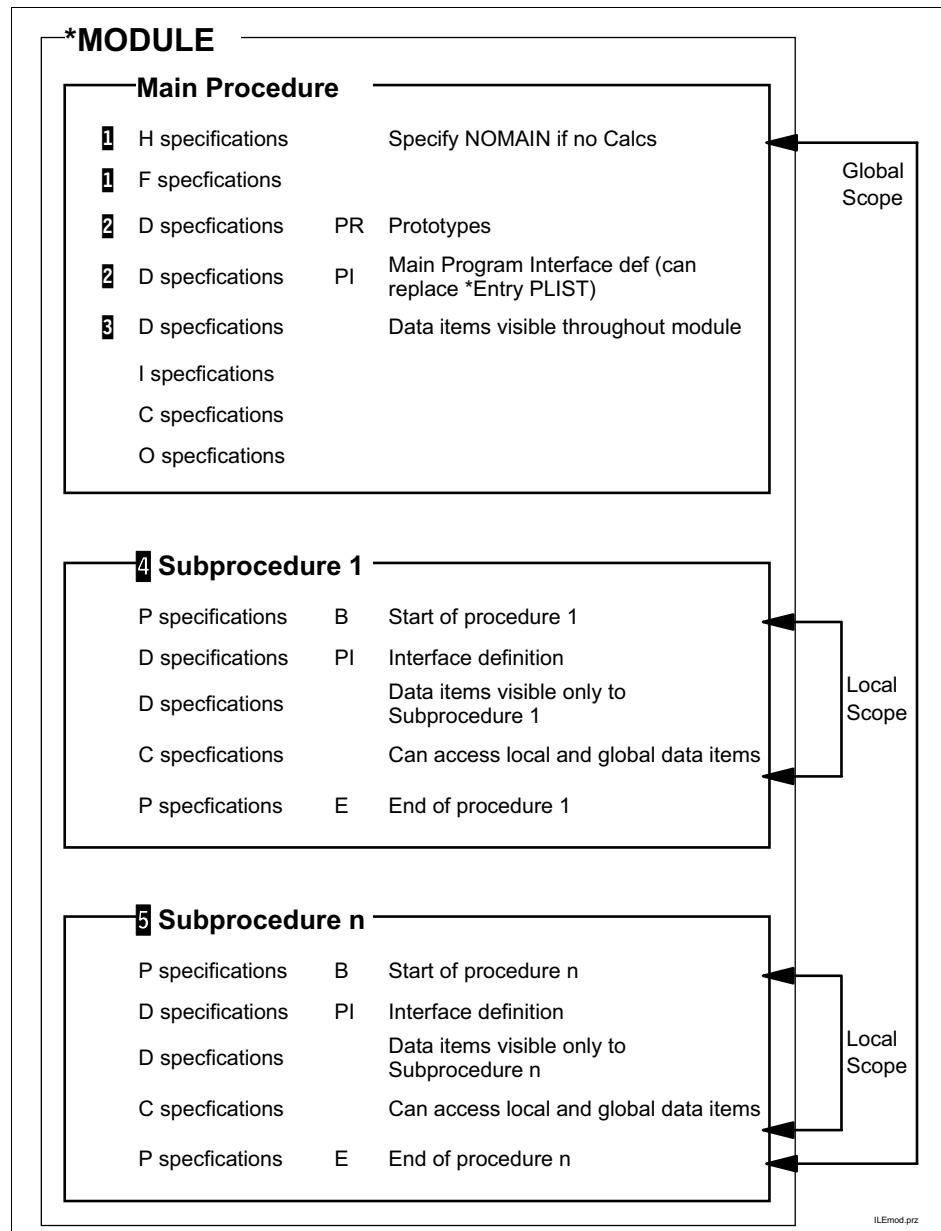


Figure 3-1 Order of coding the source elements

The markers 1 through 5 shown in Figure 3-1 are defined as follows:

- 1 The NOMAIN keyword on the control spec is used if there is no mainline logic in this module. That is to say that the only calculation specs are those in the subprocedures.

Note: The file specs can now go either at the beginning of the member, just after the control spec or inside the procedure. File specs after the control spec can be used either by the mainline code and/or by the subprocedures. If the file spec is coded inside a procedure it is available only within that procedure.

- 2 The prototypes always go before the first procedure in the module. Any global variables also go before the first procedure.

- 3** The procedure specification and interface definition for the main procedure should be at the top of the module. The main program procedure is usually noted by an EXTPROC or EXTPGM keyword on the prototype. This is used by the compiler and binder to determine how to handle the module and how other programs can call this one. This is discussed in more detail in Chapter 4, “An ILE guide for the RPG programmer” on page 77.

Tip: Notice that you should not be coding I specs anymore. There is no free form code for input specs.

- 4** After the calculation specs, the procedure is ended with an end procedure statement. The name of the procedure is optional on this line. However, it is considered good programming style to include it for subprocedures.

Tip: There is also no way to code output specs in free form.

- 5** Any other subprocedures would follow, each with its own set of procedure, data, and calc specs.

The procedure-interface definition can be placed anywhere within the definition specs. However, it is considered a good programming practice that you code it immediately following the procedure spec. The use of a RETURN opcode is recommended but does not need to be coded unless a value is to be returned. The subprocedure automatically returns when it reaches the end procedure spec.

3.3.4 Calling your subprocedures

Always use a prototyped call to your subprocedures. It can be done with a CALLB, but why would you? The prototyped call provides a much better interface and offers the benefits of parameter checking and a number of other advantages that are discussed in 3.6.1, “The power of prototyping” on page 66. For these reasons, this section only discusses the use of prototyped subprocedure calls.

In RPG, prototyped calls are also known as *free-form calls*. A free-form call refers to the fact that the arguments for the call are specified using free-form syntax, much like the arguments for built-in functions.

How you call the procedure depends on whether it has a return value or not:

- ▶ If there is no return value, the code is on its own line in the member. There is no need to use the CALLP operation.
- ▶ If there is a return value, the prototyped procedure can be in an expression. It can go anywhere an RPG BIF can go. If you do not want to use the returned value, you can code it on a line by itself and the return value is ignored.

Using either type of procedure call, you can call:

- ▶ A procedure in a separate module within the same ILE program or service program.
- ▶ A procedure in a separate ILE service program.
- ▶ A procedure in another program through the use of a procedure pointer as discussed in 3.6.3, “Using procedure pointer calls” on page 72“.

The following are some examples of using the subprocedure AFunction which was introduced in 3.3.1, “Subprocedure definition” on page 48. Notice that the value returned can be used in ways other than simply assigning it to a variable. As shown in the second example, it can also be used directly in an expression.

```
// Return a result
theResult = AFunction(aParm1 : aParm2);

// Use in an 'if' statement
if AFunction(parm1 : parm2) = 1;

// Use with a 'do until' operation code
dou AFunction(parm1 : parm2) = 1;
```

If the subprocedure does not return a value or you want to ignore the return value, you can use the following syntax to call it:

```
// Use with no result returned
AFunction(aParm1 : aParm2);
```

3.4 Moving from subroutines to subprocedures

Existing subroutines often make good candidates for subprocedures. This section shows you how to take a subroutine and convert it into a subprocedure.

3.4.1 Why use subprocedures

As noted in 3.2, “Advantages of using subprocedures” on page 47, subprocedures offer a number of features that are not available with subroutines. Nonetheless, if you do not require the improvements offered by subprocedures, you can continue use a subroutine. The processing of a subroutine is still slightly faster than a bound call to a subprocedure.

3.4.2 Subroutine example DATESUBR

The program example shown in Example 3-2 is called with a single date field in *ISO format as the single input parameter. Based on this date, it calculates the day of the week and displays it to the user. To be clear, this program uses subroutines to accomplish its work. Later, in 3.4.4, “DATEMAIN1 subprocedure example” on page 43, you will see the same code re-written with subprocedures.

Example 3-2 Subroutine example DATESUBR

```
// DATESUBR from SUBPROCSRC
// This routine uses Sunday as day 7 - any date that represents
// a Sunday will work
Dcl-c AnySunday const(D'2015-12-27');

Dcl-s WorkNum zoned(7 : 0);
Dcl-s WorkDay zoned(1 : 0);
Dcl-s DayName char(9);

// Days of the week name table - note field names are required
Dcl-ds NameData;
d1 char(9) Inz('Monday');
```

```

d2 char(9) Inz('Tuesday');
d3 char(9) Inz('Wednesday');
d4 char(9) Inz('Thursday');
d5 char(9) Inz('Friday');
d6 char(9) Inz('Saturday');
d7 char(9) Inz('Sunday');

// Define the array as an overlay of the DS name
Name char(9) Dim(7) pos(1);
end-Ds;

// Program parameters
dcl-pi datesubr;
  workdate date(*iso);
end-Pi;

// Call DayOfWeek subroutine to initialize field Workday
Exsr DayOfWeek;

// Using Workday, initialize DayName from name table
DayName = Name(WorkDay);

// Display resulting name
Dsply DayName;

// Terminate Program
*InLR = *On;

// Subroutine: DayOfWeek (Day of the Week)
// Using the content of WorkDate, will initialize the field
// WorkDay with a number representing the day of the week
// (Monday = 1, ... , Sunday = 7)
Begsr DayOfWeek;
  WorkNum = %diff( WorkDate : Anysunday : *days);
  WorkDay = %rem(WorkNum :7);

  // Testing for < 1 allows for the situation where the input date
  // is earlier than the base date (AnySunday)
  If WorkDay < 1;
    WorkDay += 7;
  Endif;
Endsr;

```

Try it yourself: You can try the example shown in Example 3-2 on page 53 by compiling the code from this section on your IBM i system. You can use the following command to create the program:

```
CRTBNDRPG PGM(rpgiscool/datesubr) SRCFILE(rpgiscool/subprocsrc)
```

Then, use the following command to execute it:

```
CALL PGM(rpgiscool/datesubr) PARM('1969-01-31')
```

3.4.3 Transforming a subroutine to a subprocedure

There are four steps required to transform a subroutine into a subprocedure. These are explained in this section:

1. Remove the BEGSR/ENDSR instructions, and specify the return value.
2. Add the begin and end procedure specs and the procedure interface.
3. Define the prototype.
4. Replace the EXSR instruction in the main procedure with the subprocedure invocation.

The following modifications can also be made in order to use the language more efficiently:

- ▶ Use a procedure interface for the Main procedure's parameters. For more information, see 3.5.2, "Using the MAIN control option keyword and a prototype for the main procedure" on page 60.
- ▶ Use other subprocedures in your subprocedure. For more information, see 3.5.3, "Subprocedures using subprocedures" on page 61.
- ▶ Place your subprocedures in a Service Program. For more information, see 3.5.4, "Using an ILE service program" on page 62.

Note: In this section, the markers 1 through 5 next to the code snippets in the examples correspond to the same numbers in the full program listing found in 3.4.4, "DATEMAIN1 subprocedure example" on page 57.

Tip: Section 2.1, "The essential RPG IV style guide" on page 19, offers you some suggestions on the proper coding style for subprocedures.

Step 1: Returning a value

This first step focuses on removing the subroutine identifier and specifying a return value instruction:

1. Using the subroutine code as a base, remove the BEGSR (Begin subroutine) and ENDSR (End Subroutine) opcodes.
2. Add a RETURN instruction and specify the return value in factor 2, as shown with marker 1 in the following code sample.

This specifies the value that is returned to the caller. This can be a simple value.

```
// Testing for < 1 allows for the situation where the input date
// is earlier than the base date (AnySunday)
If WorkDay < 1;
  WorkDay += 7;
Endif;
return workday;      1
```

It can also be an expression since the RETURN operation code is a free-form operation:

```
// Testing for < 1 allows for the situation where the input date
// is earlier than the base date (AnySunday)
If WorkDay < 1;
  return WorkDay + 7;  1
else;
  return workDay;
Endif;
```

Step 2: Defining the interface

This section defines the interface that defines the entry parameters and the return value of the subprocedure:

1. Add the declare procedure and end procedure statements

Subprocedures begin with a declare procedure and end with an end procedure spec. The declare procedure spec names the subprocedure and has a similar layout to the data spec. Note that the end procedure spec does not require that the name of the procedure be present.

```
// SubProcedure: DayOfWeek (Day of the Week)
// The subprocedure accepts a valid date (format *ISO) and returns
// a number (1 digit) representing the day of the week
// (Monday = 1, ..., Sunday = 7)

dcl-Proc DayOfWeek export;          2
...
end-Proc DayOfWeek;
```

2. Declare the Procedure Interface. Now, you need a Procedure Interface (PI). The first line of the PI defines the data type and size of the return value. In this example, this is a single digit numeric field with no decimal places.

Subsequent lines define the parameters passed to or from the subprocedure. This example has only one parameter, the field WorkDate, which is an *ISO format date.

The PI is typically the first D spec in the subprocedure and effectively acts as the *ENTRY PLIST. The end of the parameter list is indicated by an end procedure interface statement.

```
// SubProcedure definition: DayOfWeek           3
Dcl-pi DayOfWeek zoned(1 : 0);
      workDate date(*iso);
end-Pi;
```

For the purpose of this example, the field definitions and constants, which are used only by the subprocedure, were moved into the subprocedure itself. These local data items are now be accessible only within the subprocedure.

Returning values subprocedures: A subprocedure can return, at most, one value. If you need to return more than one value, you can choose any of the following options:

- ▶ Return a data structure.
- ▶ Return a pointer to a data structure.
- ▶ Modify the content of parameters passed to you. This is only possible if the parameter was passed by reference. For more information, see “Passing by reference” on page 70.
- ▶ Use ability of ILE to share data items between modules via the IMPORT or EXPORT keyword. For more information, see Chapter 4, “An ILE guide for the RPG programmer” on page 77.

Step 3: Defining the prototype

The format of the parameters in the prototype must match the Procedure Interface (PI). The parameters do not need to be named. If they are named, the name does not need to match the one specified on the PI. In fact, the compiler is going to completely ignore the name on the parameter. It is only interested in its data type and size.

```
// Prototype for subprocedure DayOfWeek      4
Dcl-pr DayOfWeek zoned(1 : 0);
  InputDate date(*ISO);
end-Pr;
```

Step 4: Calling the subprocedure

The last step is to replace the Exit subroutine EXSR instructions with an invocation of the subprocedure. Since we want to save the value our subprocedure returns, we need to use it in an expression.

Most RPG programmers would probably start out by coding this way:

```
dcl-s DayNo      zoned(1 : 0);
. . .
// Call dayOfWeek subprocedure, passing Workdate, will return DayNo
dayNo = DayOfWeek(WorkDate);
// Use DayNo as an index to table Name to derive DayName value
DayName = Name(dayNo);
```

However, there is a better way. Remember that subprocedures can appear anywhere in an expression where a variable of the same type can be used. Here is the same example with a cleaner programming style:

```
// Using subprocedure DayofWeek, initialize DayName with table Name      5
DayName = Name(DayOfWeek(WorkDate));
```

The compiler generates the necessary call to the DayOfWeek procedure and uses the returned value to supply the array subscript for the Name array.

3.4.4 DATEMAIN1 subprocedure example

Example 3-3 shows the complete result code. This program now contains a subprocedure that basically has the same functionality as the subroutine shown in 3.4.2, “Subroutine example DATESUBR” on page 53. As described in the steps in 3.4.3, “Transforming a subroutine to a subprocedure” on page 55, these steps enhance this code to take advantage of the power of using subprocedures efficiently in RPG IV.

Example 3-3 DATEMAIN1 subprocedure example

```
// DATEMAIN1 from SUBPROCSRC

// Prototype for subprocedure DayOfWeek      4
Dcl-pr DayOfWeek zoned(1 : 0);
  InputDate date(*ISO);
end-Pr;

// Days of the week name table - note field names are required
Dcl-ds NameData;

  d1 char(9) Inz('Monday');
  d2 char(9) Inz('Tuesday');
```

```

d3 char(9) Inz('Wednesday');
d4 char(9) Inz('Thursday');
d5 char(9) Inz('Friday');
d6 char(9) Inz('Saturday');
d7 char(9) Inz('Sunday');

// Define the array as an overlay of the DS name
Name char(9) Dim(7) pos(1);
end-Ds;

dcl-s DayName char(9);

Dcl-pi DateMain1;
WorkDate date(*ISO);
end-Pi;
// Using subprocedure DayofWeek, initialize DayName with table Name
DayName = Name(DayOfWeek(WorkDate));      5

// DISPLAY RESULTING NAME
Dsply DayName;

// Terminate Program
*InLR = *On;

return;
// SubProcedure: DayOfWeek (Day of the Week)
// The subprocedure accepts a valid date (format *ISO) and returns
// a number (1 digit) representing the day of the week
// (Monday = 1, ... , Sunday = 7)

dcl-Proc DayOfWeek;                      2

Dcl-pi DayOfWeek zoned(1 : 0);           3
workDate date(*iso);
end-Pi;

// Constant defined here as only place it is used
Dcl-c AnySunday const(D'2015-12-27');

// Local variables
Dcl-s WorkNum zoned(7 : 0);
Dcl-s WorkDay zoned(1 : 0);

WorkNum = %diff( WorkDate : AnySunday : *days);
WorkDay = %rem(WorkNum :7);

// Testing for < 1 allows for the situation where the input date
// is earlier than the base date (AnySunday)
If WorkDay < 1;
  WorkDay += 7;
Endif;

return workday;                         1
end-Proc DayOfWeek;                     2

```

Try it yourself: You can try Example 3-3 on page 57 by compiling the code from this section on your IBM i system. You can use the following command to create the program:

```
CRTBNDRPG PGM(rpgiscool/datemain1) SRCFILE(rpgiscool/subprocsrc)
DFTACTGRP(*NO) ACTGRP(*NEW)
```

Then, use the following command to execute it:

```
CALL PGM(rpgiscool/datemain1) PARM('1972-03-08')
```

3.5 Using subprocedures efficiently

This section enhances the example shown in 3.4.4, “DATEMAIN1 subprocedure example” on page 57 by:

- ▶ Using /COPY members for prototypes.
- ▶ Using the MAIN control option keyword and using a prototype for the main procedure.
- ▶ Using a subprocedure from within a subprocedure.
- ▶ Creating a Service Program from a subprocedure.

These techniques, among others, show the power of using subprocedures in your application design.

3.5.1 Using /COPY members for prototypes

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 placed in a separate module (source member). It is critical that the prototype in the calling procedure match the one in the defining procedure, since it is the one in the module containing the subprocedure that the compiler verified for you.

Prototypes for groups of related functions should be placed in a single member, for example, date routines, validation routines, and so on. You can also choose to group prototypes per service program.

Important: There is no performance penalty for unused prototypes.

It is important to note that /COPY, does not mean that you should copy the source lines using Source Edit Utility (SEU) (or any other editor for that matter). One of the objectives of using prototypes is to avoid making mistakes when calling programs and procedures. If the prototype exists in each individual source member, it can be edited in each member with a resulting loss of integrity.

The examples shown in the following sections list a prototype in their specific members and include them in the main source section when required. The /COPY instruction can be used as follows:

```
/COPY library/sourcefile,member
```

For more information, see the *Programming IBM Rational Development Studio for i, ILE RPG Reference* manual:

https://www.ibm.com/support/knowledgecenter/api/content/nl/en-us/ssw_ibm_i_72/rzasd/sc092508.pdf

3.5.2 Using the MAIN control option keyword and a prototype for the main procedure

You can use prototypes to validate the parameters on any kind of call, not just bound calls or subprocedure calls.

For the example used in this chapter, the formal *ENTRY PLIST has already been replaced with a procedure interface. Now we want to use a prototype as well. As we are not going to be using the RPG cycle in this program, we want to use the main keyword to eliminate the cycle from the generated code. The resulting prototypes can be used by other programs. We realize that, given the nature of this sample program, this is not likely, but the principle sounds good.

Note: The markers **A** through **H** are used throughout the following sections to identify matching source code examples up to, but not including, 3.6.3, “Using procedure pointer calls” on page 72“.

For this example, the main procedure’s input parameters would be coded as the following:

```
// Program input parameter          F
Dcl-pi myMainModule;
  WorkDate date(*ISO);
end-Pi;
```

Of course, you also have to include a prototype with the same information into your main procedure. In this case, you would use a /COPY member since other programs might want to use a prototyped call to invoke the main program. The name of the prototype does not need to match the name of your module if you specify the EXTPGM keyword on the prototype definition. Note that one advantage of using prototypes for this type of call is that the actual program or procedure name can be “overridden” to a more meaningful name.

```
// Prototype for main program        G
dcl-pr myMainModule extpgm('DATEMAIN2');
  workdate date(*iso);
end-Pr;
```

Coding considerations: A main procedure is always exported, which means that other procedures in the program can call the main procedure by using bound calls. The call interface of a main procedure can be defined in one of two ways:

- ▶ Using a prototype and procedure interface.
- ▶ Using an *ENTRY PLIST without a prototype.

However, a prototyped call interface is much more robust since it provides parameter checking at compile time. If you prototype the main procedure, you also dictate how it is to be called. This is achieved by specifying either the EXTPROC or EXTPGM keyword on the prototype definition:

- ▶ If EXTPGM is specified, then an external program call is used.
- ▶ If EXTPROC is specified, it is called by using a bound procedure call.
- ▶ If neither keyword is specified, the compiler defaults to EXTPROC.

For more information on these keywords, see “External naming” on page 69. Note that it is not necessary that the called program uses a procedure interface for the calling program to use a prototype. These options can be mixed with more traditional *ENTRY PLISTS and CALL/PARM operations.

3.5.3 Subprocedures using subprocedures

Since the ability to obtain a day name is useful to other programs in the example used in this chapter, the function is extracted as a subprocedure. Because this function requires the input date to be converted into a day number, it invokes the DayOfWeek function defined previously. The new subprocedure NameOfDay would be defined as shown in Example 3-4.

Example 3-4 NameOfDay subprocedure

```
// SubProcedure: NameOfDay (Name of the Day)
// The subprocedure accept a valid date (format *ISO) and return
// a string representing the name of the day
dcl-Proc NameOfDay  Export;          B

Dcl-pi NameOfDay char(9);
  WorkDate date(*ISO);
end-Pi;
// Days of the week name table - note field names are required
Dcl-ds NameData;
  d1 char(9) Inz('Monday');
  d2 char(9) Inz('Tuesday');
  d3 char(9) Inz('Wednesday');
  d4 char(9) Inz('Thursday');
  d5 char(9) Inz('Friday');
  d6 char(9) Inz('Saturday');
  d7 char(9) Inz('Sunday');

// Define the array as an overlay of the DS name
  Name  char(9) Dim(7) pos(1);
end-Ds;
Return Name(DayOfWeek(Workdate));
end-Proc NameOfDay;
```

Here is the prototype that is related to the NameOfDay subprocedure. As noted previously, this prototype must be included in the calling procedure, and in the subprocedure itself, unless the subprocedure is part of the same module as the caller.

```
// Prototype for subprocedure DayName
Dcl-pr NameOfDay char(9);
  InputDate date(*ISO);
end-Pr;
```

D

From the main procedure, the new procedure is invoked by this instruction:

```
// Using subprocedure DayName, Retrieve the Name of the day from
// WorkDate
DayName = NameOfDay(WorkDate);
```

E

An exercise for you

A different approach to the last example would be to leave the two subprocedures independent from each other and to use them together on the same EVAL statement in the main Procedure, as follows:

- * Using the Day number returned by subprocedure DayOfWeek from the
- * WorkDate, retrieve the Day Name from subprocedure NameOfDay

C Eval DayName = NameOfDay(DayOfWeek(WorkDate))

To use the return value of a subprocedure as the input value of another one, on the same free-form expression statement, you must specify the input parameter of the second procedure as passed by the value.

Example 3-5 shows what the NameOfDay subprocedure would look like using the VALUE keyword on the parameter definition. Do not forget to change the prototype. More information on passing parameters to subprocedures can be found in “[Parameter passing styles](#)”.

Example 3-5 NameOfDay subprocedure using the VALUE keyword on the parameter definition

```
// SubProcedure: NameOfDay (Name of the Day)
// The subprocedure accept a day number (monday = 1,..., Sunday = 7)
// and returns a string representing the name of the day
dcl-Proc NameOfDay  Export;

Dcl-pi NameOfDay char(9);
  WorkDay zoned(1 : 0) value;
end-Pi;

// Days of the week name table - note field names are required
Dcl-ds NameData;
  d1 char(9) Inz('Monday');
  d2 char(9) Inz('Tuesday');
  d3 char(9) Inz('Wednesday');
  d4 char(9) Inz('Thursday');
  d5 char(9) Inz('Friday');
  d6 char(9) Inz('Saturday');
  d7 char(9) Inz('Sunday');

// Define the array as an overlay of the DS name
Name  char(9) Dim(7) pos(1);
end-Ds;
Return Name(Workday);
end-Proc NameOfDay;
```

3.5.4 Using an ILE service program

Since these two subprocedures might be useful in many other programs that use dates, it would be a good idea to compile them into a separate module. In fact, you may want to use them in an ILE service program, but the same principles apply even if you want to simply bind the resulting module by copy. In the following examples, you will see what you need to add to these subprocedures to compile them as a separate module.

The NOMAIN keyword is used on the control option spec since this allows you to take advantage of the cycleless feature of RPG IV that improves performance (see marker [H](#)). Perhaps even more useful is the fact that by coding NOMAIN, the compiler is stopped from complaining that it cannot determine how the program ends.

NOMAIN: The NOMAIN keyword is not required, but causes slightly smaller modules. NOMAIN tells the compiler not to include any RPG cycle logic in this module.

The keyword is only allowed if and when there are no calc specs in the source member prior to the first subprocedure. In other words, NOMAIN can only be used if there is no main procedure logic coded in this module.

If you specify NOMAIN, you cannot use the **CRTBNDRPG** command on the source member. This is because the **CRTBNDRPG** command requires that the module contain a program entry procedure. Only a main procedure can be a program entry procedure.

Similarly, when using the **CRTPGM** command to create a program, keep in mind that a NOMAIN module cannot be an entry module since it does not have a program entry procedure.

A subprocedure can be exported, allowing it to be called from other modules. To indicate that it is to be exported, you must specify the keyword EXPORT on the Procedure Begin spec. If EXPORT is not specified, the subprocedure can only be called from other procedures within the module.

When the subprocedures are moved to a different module, they can no longer see the prototype definition placed in the main procedure. You must include a copy of the prototype definition in the subprocedures module. The fact that these prototypes are required in multiple places is the reason it is considered a best practice to use the /COPY directive to bring in the prototype code. This way you can ensure the correct prototype is always used.

Using the preceding example, Example 3-6 shows what the new service program module would look like.

Example 3-6 New service program module

```
// DATESRVPG2 from SUBPROCSRC

// Control option spec, specify no main procedure in this code      H
ctl-opt Nomain;

// Include prototypes          D
/Copy Radredbook/SUBPROCSRC,DATESUBPR2

// SubProcedure: NameOfDay (Name of the Day)
// The subprocedure accept a valid date (format *ISO) and return
// a string representing the name of the day                      B
dcl-Proc NameOfDay  Export;

// Procedure interface definition          C
Dcl-pi NameOfDay char(9);
  WorkDate date(*ISO);
end-Pi;
// Days of the week name table - note field names are required
Dcl-ds NameData;
d1 char(9) Inz('Monday');
d2 char(9) Inz('Tuesday');
d3 char(9) Inz('Wednesday');
d4 char(9) Inz('Thursday');
```

```

d5 char(9) Inz('Friday');
d6 char(9) Inz('Saturday');
d7 char(9) Inz('Sunday');

// Define the array as an overlay of the DS name
Name  char(9) Dim(7) pos(1);
end-Ds;

// Returning name from nameData table data using
// DayOfWeek subprocedure
Return Name(DayOfWeek(Workdate)); E

end-Proc NameOfDay;

// SubProcedure: DayOfWeek (Day of the Week)
// The subprocedure accepts a valid date (format *ISO) and returns
// a number (1 digit) representing the day of the week
// (Monday = 1, ... , Sunday = 7) B
dcl-Proc DayOfWeek export;
// Procedure interface definition: DayOfWeek
Dcl-pi DayOfWeek zoned(1 : 0);
    workDate date(*iso);
end-Pi; C

// Constant defined here as only place it is used
Dcl-c AnySunday const(D'2015-12-27');

// Local variables
Dcl-s WorkNum zoned(7 : 0);
Dcl-s WorkDay zoned(1 : 0);

WorkNum = %diff( WorkDate : AnySunday : *days);
WorkDay = %rem(WorkNum :7); A

// Testing for < 1 allows for the situation where the input date
// is earlier than the base date (AnySunday)
If WorkDay < 1;
    return WorkDay + 7;
else;
    return workDay;
Endif;
end-Proc DayOfWeek;

```

In this example, the subprocedure DayOfWeek does not require the EXPORT keyword since it is currently only used by the subprocedure NameOfDay, which is located in the same module. We have chosen to export it so that we can make it available to all of our programmers. The main procedure would now look like what is shown in Example 3-7.

Example 3-7 Main procedure

```

// DATEMAIN2 from SUBPROCSRC
ctl-opt main(myMainModule);

// Include prototype for Main procedure
/Copy SUBPROCSRC,DATEMAINP2 G

```

```

// Include prototypes for subprocedures DayOfWeek and NameOfDay
/Copy SUBPROCSRC,DATESUBPR2

// Stand Alone Fields
Dcl-s DayName char(9);

// Program start
dcl-proc myMainModule;
  // Program input parameter
  Dcl-pi myMainModule;
    WorkDate date(*ISO);
  end-Pi;

  // Using subprocedure DayName, Retrieve the Name of the day from
  // WorkDate
  DayName = NameOfDay(WorkDate); E

  // Display the content of DayName
  Dsply DayName;

  // Terminate Program
  return;
end-Proc;

```

You do not want to forget to code the prototypes. The code shown in Example 3-8 is for the main procedure.

Example 3-8 Coding the prototype in the main procedure

```

// DATEMAINPR from SUBPROCSRC

// Prototype for main program
dcl-pr myMainModule extpgm('DATEMAIN2'); G
  workdate date(*iso);
end-Pr;

```

Example 3-9 shows the code subprocedures:

Example 3-9 Subprocedures code

```

// DATESUBPR from SUBPROCSRC

// Prototype for subprocedure DayOfWeek
Dcl-pr DayOfWeek zoned(1 : 0);
  InputDate date(*ISO);
end-Pr; D

// Prototype for subprocedure NameOfDay
Dcl-pr NameOfDay char(9);
  InputDate date(*ISO);
end-Pr; D

```

Try it yourself: To recreate the example shown in this section on your system, you need to compile the two modules using the following commands:

- ▶ For the service program (subprocedures module), use:

```
CRTRPGMOD MODULE(rpgiscool/datesrvpg2) SRCFILE(rpgiscool/subprocsrc)
CRTSRVPGM SRVPGM(rpgiscool/datesrvpg2)MODULE(rpgiscool/datesrvpg2)
EXPORT(*ALL)
```

- ▶ For the main procedure, use:

```
CRTRPGMOD MODULE(rpgiscool/datemain2) SRCFILE(rpgiscool/subprocsrc)
CRTPGM PGM(rpgiscool/datemain2) MODULE(rpgiscool/datemain2)
BNDSRVPGM(rpgiscool/datesrvpg2)
```

Then, use the following command to execute it:

```
CALL PGM(rpgiscool/datemain2) PARM('1998-05-28')
```

3.6 More on subprocedures

This section shows more features provided by the optional prototype keywords and different ways of calling subprocedures.

3.6.1 The power of prototyping

Prototypes can do much more for you than simply defining the parameters to subprocedures. By using some of the optional keywords for prototypes, you can have the compiler check the number and type of parameters in your calling program and procedures. It can even accommodate small types of mismatches, such as passing an integer when the callee expects a value with decimal places, for example see Example 3-10.

Example 3-10 Passing an integer when the callee expects a value with decimal places

```
Dcl-pr OvrDBFile ExtPgm('QCMDEXC');
  CmdString  char(3000)      Options(*Varsize)
              Const;
  CmdLength  packed(15 : 5)  Const;
  CmdOpt     char(3)        Options(*NoPass)
              Const;
end-Pr;

Dcl-s CustMast   char(100);
Dcl-c Override1  'OVRDBF FILE(';
Dcl-c Override2  ') TOFILE(';
Dcl-c Override3  ') SHARE(*YES)';

CustMast = Override1 + 'CustMast' +
           Override2 + 'MyLibrary/CustMast' +
           Override3;

OvrDBFile( CustMast : %Len(CustMast));
*InLR = *On;
```

The use of the CONST keyword allows the compiler to accommodate a mismatch in the definition of the parameters between the callee and the caller. For example, this might happen when the callee expects a packed decimal value of five digits with no decimal places and the caller wants to pass a three digit signed numeric. Normally you would have to create a temporary variable (packed - five digits), move the three digit number to it, and then pass the temporary field as the parameter. When you use the CONST keyword, you are specifying that it is acceptable that the compiler make a copy of the data prior to sending it, if necessary, to accommodate these mismatches.

Another benefit of using CONST is that it also allows an expression to be passed as a parameter. For more information, see 3.6.2, “Parameter passing styles” on page 70.

The use of the option *NOPASS on the OPTIONS keyword means the parameter does not have to be passed on the call. Any parameters following that spec must also have *NOPASS specified. When the parameter is not passed to a program or procedure, the called program or procedure simply functions as if the parameter list did not include that parameter. When parameters are not mandatory, you can also use the option *OMIT, which indicates the value *OMIT is allowed for that parameter when calling the subprocedure. *OMIT is only allowed for CONST parameters and parameters that are passed by reference.

Other options on the OPTIONS keyword are *VARSIZE, *STRING, and *RIGHTADJ. For more information on these keywords, see the *Programming IBM Rational Development Studio for i, ILE RPG Reference* manual:

https://www.ibm.com/support/knowledgecenter/api/content/nl/en-us/ssw_ibm_i_72/rzasd/sc092508.pdf

Converting the date format

In this example, the CONST and DATFMT keywords are combined. As a result, the compiler can generate a temporary (hidden) date field in the calling program or procedure, if necessary, to convert the date format used in the caller to the format used in the called subprocedure. This example is based on the example used in 3.4, “Moving from subroutines to subprocedures” on page 53 where the date format *ISO is used on all the dates definitions.

However, suppose that you wanted to use the DayOfWeek procedure from a program that defines its date fields as having the *USA format. All you need to do is modify the date subprocedures so that they include the CONST keyword on the parameter definitions in the procedure interface and prototypes. Once you recompile the subprocedures and rebuild the service program, you can safely call using the *USA date field.

Example 3-11 show how the modified prototypes look.

Example 3-11 Modified prototypes

```
// DATESUBPR3 from SUBPROCSRC

// Prototype for subprocedure DayOfWeek
Dcl-pr DayOfWeek zoned(1 : 0);
    InputDate  date(*ISO) const;
end-Pr;

// Prototype for subprocedure DayName
Dcl-pr NameOfDay char(9);
    InputDate  date(*ISO) const;
end-Pr;
```

You also need to modify the procedure interface (PI) of those two subroutines to reflect the changes made in the prototype as shown in Example 3-12.

Note: The full version of the code can be seen in 3.5.4, “Using an ILE service program” on page 62.

Example 3-12 Modifying the procedure interface

```
// DATESRVPG3 from SUBPROCSRC

// H
ctl-opt Nomain;

// D
// Include prototypes
/Copy SUBPROCSRC,DATESUBPR3

// SubProcedure: NameOfDay (Name of the Day)
// The subprocedure accept a valid date (format *ISO) and return
// a string representing the name of the day B
dcl-Proc NameOfDay Export;

// C
// Procedure interface definition
Dcl-pi NameOfDay char(9);
  WorkDate date(*ISO) const;
end-Pi;

// SubProcedure: DayOfWeek (Day of the Week)
// The subprocedure accepts a valid date (format *ISO) and returns
// a number (1 digit) representing the day of the week
// (Monday = 1, ..., Sunday = 7) B
dcl-Proc DayOfWeek export;
  // Procedure interface
  Dcl-pi DayOfWeek zoned(1 : 0);
    workDate date(*iso) const;
  end-Pi;
```

Example 3-13 shows the main procedure modified to use a *USA date format.

*Example 3-13 Main procedure modified to use a *USA date format*

```
// DATEMAIN3 from SUBPROCSRC

ctl-opt main(datemain3);

// G
// Include prototype for Main procedure
/Copy SUBPROCSRC,DATEMAINP3

// Include prototypes for subprocedures DayOfWeek and NameOfDay
/Copy SUBPROCSRC,DATESUBPR3

// Stand Alone Fields
Dcl-s DayName char(9);

// Program start
dcl-proc dateMain3;
```

```

// Program input parameter
Dcl-pi DateMain3;
  WorkDate date(*USA);
end-Pi;

// Using subprocedure DayName, Retrieve the Name of the day from
// WorkDate
DayName = NameOfDay(WorkDate);

// Display the content of DayName
Dsply DayName;

// Terminate Program
*InLR = *On;
end-Proc;

```

You also need to modified the prototype of the main procedure as shown in Example 3-14.

Example 3-14 Modifying the prototype of the main procedure

```

// DATEMAINP3 from SUBPROCSRC in RPGISCOOL
// Prototype for main program
dcl-pr datemain3 extpgm('DATEMAIN3');
  workdate date(*usa);
end-Pr;

```

Try it yourself: To recreate this example on your system, you need to compile the two modules using the following commands:

- ▶ For the service program (subprocedures module), use:

```

CRTRPGMOD MODULE(rpgiscool/datesrvpg3) SRCFILE(rpgiscool/subprocsrc)
CRTSRVPGM SRVPGM(rpgiscool/datesrvpg3) MODULE(rpgiscool/datesrvpg3)
EXPORT(*ALL)

```

- ▶ For the main procedure, use:

```

CRTRPGMOD MODULE(rpgiscool/datemain3) SRCFILE(rpgiscool/subprocsrc)
CRTPGM PGM(rpgiscool/datemain3)MODULE(rpgiscool/datemain3)
BNDSRVPGM(rpgiscool/datesrvpg3)

```

Then, use the following command to execute it:

```
CALL PGM(rpgiscool/datemain3) PARM('05/28/2000')
```

External naming

Another feature of prototypes is the use of the EXTPGM and EXTPROC keywords. ILE procedure names can be up to 128-bytes long and can be of mixed case. Typically, mixed case names are used only in procedures written in C. However, since you can use prototypes to call C functions, it is important to understand the use of the EXTPROC keyword to accommodate the mixed case names typical of C functions.

If the keyword EXTPGM or EXTPROC is specified on the prototype definition, calls use the procedure name given, but the actual procedure or program called is the one specified in the EXTPGM or EXTPROC keyword. If neither keyword is specified, then the external name is the prototype name, which is the name specified on the Prototype definition and converted to uppercase. See Example 3-15.

F

E

G

Example 3-15 Using the EXTPGM and EXTPROC keywords

```
dcl-pr myProcName    extproc('A_really_long_name()');
  procParm1 packed(10 : 2);
end-Pr;

dcl-pr AProgName    extpgm('QODBNAM');
  parm1      int(10);
end-Pr;

myProcName(Procparm1);

AProgName(Parm1);
```

3.6.2 Parameter passing styles

A case of mixed case: It is possible to create an RPG IV subprocedure that exports a mixed-case name. You might need to do this if, for example, you are writing a user exit point for a system designed with C programmers in mind, or if you want to convince people that you wrote all of your routines in C.

All you need to do is to code the EXTPROC keyword on the prototype for your subprocedure. The compiler exports the name in mixed case as specified. For example:

```
// This prototype will export the name 'MixedCase'
dcl-pr MixedCase    extproc('MixedCase');

// This one will export the name 'MIXEDCASE'
dcl-pr MixedCase;
```

Program calls, including system API calls, require that parameters be passed by reference. However, there is no such requirement for procedure calls. ILE RPG allows three methods for passing and receiving prototyped parameters:

- ▶ By reference (the default).
- ▶ By value (keyword VALUE on the parameter definition).
- ▶ By read-only reference (keyword CONST on the parameter definition).

Parameters that are not prototyped can only be passed by reference.

Important: IBM i program calls can only pass parameters by reference and by read-only reference, not by value. The keyword CONST should be used with every input parameter on the IBM i API prototypes definition.

Passing by reference

The default parameter passing style for RPG IV is to pass by reference. When a parameter is passed by reference, the compiler only passes a (hidden) pointer to the data value, rather than passing the actual value. Consequently, you do not have to code any keywords on the parameter definition to pass the parameter in this way. You must pass parameters by reference to a procedure when you expect the callee to modify the field passed. You might also want to pass by reference to improve run-time performance, for example, when passing large character fields.

Note that parameters that are passed on external program calls can only be passed by reference. It is not possible to pass a parameter by value to a *PGM object.

Passing by value: Keyword VALUE

With a prototyped procedure, you can pass a parameter by value instead of by reference. When a parameter is passed by value, the compiler passes the actual data to the called procedure.

Passing by value allows you to:

- ▶ Pass literals and expressions as parameters.
- ▶ Pass parameters that do not match exactly the type and length that are expected. Value parameters must match the type specified in the prototype. However, the format can be different.
- ▶ Pass a variable that, from the caller's perspective, is not modified.

When a parameter is passed by value, the called program or procedure can change the value of the parameter. However, the caller never sees the changed value.

One primary use for passing by value is that it allows for less stringent matching of the attributes of the passed parameter. For example, if the definition calls for a numeric field of type packed-decimal and a length of 5 with two decimal positions, you must still pass a numeric value. It can be any of the following options:

- ▶ A packed, zoned, or binary constant or variable, with any number of digits and number of decimal positions.
- ▶ A built-in function returning a numeric value.
- ▶ A subprocedure returning a numeric value.
- ▶ A complex numeric expression, such as:

```
2 * (Min(Length(First) + Length>Last) + 1): %size(Name))
```

Passing by read-only reference: Keyword CONST

An alternative means of passing a parameter to a prototyped procedure or program is to pass it by read-only reference. This method is also known as *constant reference*. Passing parameters this way is useful if you must pass the parameter by reference and you know that the value of the parameter is not changed during the call. For example, many system APIs have read-only parameters specifying formats or lengths.

Passing a parameter by read-only reference has many of the same advantages as passing by value. In particular, this method allows you to pass literals and expressions. However, it is important that you know that the parameter is not changed during the call.

When a parameter is passed by read-only reference, the compiler might copy the parameter to a temporary field and pass the address of the temporary field. This would happen whenever the parameter passed is not a strict match to the prototype. For example, the passed parameter is an expression or the passed parameter has a different format.

Note: If the called program or procedure is compiled using a prototype in a language that enforces the read-only reference method (either RPG IV using a prototyped procedure interface or C), the compiler prevents the parameter from being changed.

If the called program or procedure does not or cannot use a prototype, for example an RPG/400 program, the compiler cannot ensure that the parameter will not be changed. For this reason, you should exercise caution when defining prototypes using this parameter-passing method.

3.6.3 Using procedure pointer calls

Up to this point, all of the examples used in this chapter for calling subprocedures have used static procedure calls, also known as *bound calls*. It is also possible to call subprocedures using procedure pointers. Any procedure that can be called by using a static procedure call can also be called through a procedure pointer.

Procedure pointer calls provide a way to call a procedure dynamically. For example, you can pass a procedure pointer as a parameter to another procedure, which would then run the procedure that is specified in the passed parameter. You can also manipulate arrays of procedure names or addresses to dynamically route a procedure call to different procedures. If the called procedure is in the same activation group, the speed of a procedure pointer call is almost identical to that of a static procedure call.

To demonstrate the use of procedure pointer calls, we are going to revisit our date routines one more time. We are going to produce a second version of the *NameOfDay* subprocedure, which provides the name of the day in French. We will then modify our main program to ask the user if they want the results displayed in English or in French. While this may not be the most practical use of procedure pointer, hopefully it helps you to understand the basic principles involved.

Modifying the subprocedure and its prototypes

We start by modifying the prototypes to accommodate the additional subprocedure. As you can see in the revised source shown in Example 3-16, we have taken the opportunity to demonstrate how an RPG IV subprocedure can export its name in mixed-case (for more information, see 3.6.2, “Parameter passing styles” on page 70).

Example 3-16 Modifying the prototypes to accommodate the additional subprocedure

```
// DATESUBPR4 from SUBPROCSRC
// Prototype for subprocedure DayOfWeek
Dcl-pr DayOfWeek zoned(1 : 0);
    InputDate date(*ISO) const;
end-Pr;

// Prototype for subprocedure DayName
Dcl-pr NameOfDay char(9) ExtProc('NameOfDay');
    InputDate date(*ISO) const;
end-Pr;

// Prototype for subprocedure NomduJour
Dcl-pr NomDuJour char(9) ExtProc('NomDuJour');
    InputDate date(*ISO) Const;
end-Pr;
```

The new subroutine NomDuJour has an identical interface to the one for NameOfDay. This is essential since there will only be a single invocation point. Therefore, the parameter and return value should be identical. You could get some really interesting results if they were different.

NomDuJour is almost identical to NameOfDay. The only significant change, other than the name of the subroutine, is the values used for the days of the week. For this reason, we are not including the source here. You can find it with the other sources in SUBPRCSRC, member name DATESRVPG4.

A new prototype

One problem with using procedure pointers to call a subroutine is that you cannot use the same prototype to describe the interface that was used in the subroutine itself. Look at the code shown in Example 3-17 and you can see why.

Example 3-17 A new prototype

```
// DATESUBPR5 from SUBPROCSRC

// Prototype for NomduJour & NameofDay called via procedure pointer
Dcl-pr Name char(9) ExtProc(ProcToCall@);           1
    InputDate date(*ISO) const;
end-Pr;

// Constants for procedure pointers to NameOfDay and NomDuJour
Dcl-c NomDuJour@ %PAddr('NomDuJour');             2
Dcl-c NameofDay@ %PAddr('NameOfDay');
```

The markers **1** and **2** shown in Example 3-17 are defined as follows:

- 1** Notice that while this prototype uses the ExtProc keyword, as did the originals for NameOfDay and NomDuJour, the name in parentheses is the name of a procedure pointer and not the name of a specific subroutine.
- 2** The actual procedure pointers to be used are supplied by the two constants, which specify the mixed-case names that were previously arranged to have the subroutines export.

Modifying the main program

The main program needs to be modified in a number of areas to pose the English or French question and select the appropriate subroutine based on the response. The specific changes are identified in the code shown in Example 3-18.

Example 3-18 Modifying the main program to pose the English or French question

```
// DATEMAIN4 from SUBPROCSRC

ctl-opt main(datemain4);

// Include prototype for Main procedure
/Copy Radredbook/SubProcSrc,DATemainP4

// Include proc pointer prototype for NameOfDay/NomDuJour          1
/COPY Radredbook/SubProcSrc,DATESUBPR5

// Stand Alone Fields
Dcl-s DayName char(9);
```

```

// Data for English/French question - default reply to French      2
Dcl-c Question 'In English (E) ou Francais (F)';
Dcl-s Reply     char(1) Inz('F');
Dcl-s ProcToCall@ pointer(*Proc);                                // 3

// Program input parameter
dcl-proc DateMain4;

Dcl-pi DateMain4;
  WorkDate date(*usa);
end-Pi;

// Ask if the response is to be in English or French            4
Dsply Question ' ' Reply;
Select;

When Reply = 'F' or Reply = 'f';                                     // 5
  ProcToCall@ = NomDuJour@;
Other;
  ProcToCall@ = NameOfDay@;
EndS1;

// Using the appropriate subprocedure retrieve the Name of the day
DayName = Name(WorkDate);                                         // 6

// Display the content of DayName
Dsply DayName;

// Terminate Program
return;
end-Proc;

```

The markers **1** and **6** shown in Example 3-18 on page 73 are defined as follows:

- 1** Copy in the procedure pointer version of the prototypes.
- 2** This is the text for the message to the user asking them to select the language for the response. The actual question is asked at **4**. The reply value is pre-set to "F" (French). This is the value that is used if the user simply presses Enter.
- 3** This is the definition of the procedure pointer that will be used to invoke the subprocedure. It must have the same name that was used for the ExtProc keyword on the prototype.
- 4** The user is asked to select the language.
- 5** If the user responds that they want to select French, the procedure pointer for NomDuJour is loaded into ProcToCall@. If they enter any other value, the procedure pointer for NameOfDay is used.
- 6** The subprocedure is now invoked and the resulting name value is displayed to the user.

Try it yourself: You can recreate the examples shown in this section by using the following commands:

- ▶ Create a module for the procedures to be called via the procedure pointer call:

```
CRTRPGMOD SRCFILE(rpgiscool/subprocsrc) SRCMBR(datesrvpg4)
MODULE(rpgiscool/datesrvpg4)
```

- ▶ Create a module for the main program:

```
CRTRPGMOD SRCFILE(rpgiscool/subprocsrc) SRCMBR(datemainp4)
MODULE(rpgiscool/datemainp4)
CRTPGM PGM(rpgiscool/datemainp4) MODULE(datemainp4 datesrvpg4)
```

- ▶ To execute each program, enter the following command:

```
CALL rpgiscool/datemainp4 parm('07-02-99')
```

Subprocedure calls

Table 3-1 summarizes the different types of calls and the parameter options available for each type of call. An "X" in a particular cell indicates that such a combination of a call and parameter option is allowed. For example, you can use * OMIT as a parameter to a CALLB, but not to a CALL.

Table 3-1 Calls and the parameter options available for each type of call

	CALL	CALLB	CALLP ExtPgm	CALLP ExtProc	Expr.
Dynamic call	X		X		
Static/Bound call		X		X	X
Fixed format	X	X			
Free format			X	X	X
Uses prototype			X	X	X
Return value					X
Parmss by reference	X	X	X	X	X
Parms by value				X	X
*CONST				X	X
*VARSIZE			X	X	X
*OMIT		X		X	X
*NOPASS			X	X	X

Static call

A static procedure call is a call to an ILE procedure where the name of the procedure is resolved to an address during binding, therefore, the term static. As a result, run-time performance using static procedure calls is faster than run-time performance using conventional dynamic program calls.

Static calls allow operational descriptors, omitted parameters, and they extend the limit (to 399) on the number of parameters that are passed.

Exception handling in subprocedures

Exception handling within a subprocedure differs from a main procedure primarily because there is no default exception handler for subprocedures. As a result, situations where the default handler would be called for a main procedure correspond to an abnormal end of the subprocedure.



An ILE guide for the RPG programmer

This chapter illustrates essential functions of the integrated language environment (ILE). Examples are provided for the following topics:

- ▶ How to create programs and service programs from modules and other service programs.
- ▶ How to use activation groups meaningfully and avoid some inconveniences that can result from their careless use.
- ▶ How to handle errors using a condition handler program for ILE programs.

To take full advantage of ILE, you should also understand RPG IV subprocedures. These are also discussed in this chapter.

The following topics are discussed in this chapter:

- ▶ 4.1, “Introduction to ILE” on page 78
- ▶ 4.2, “ILE tips for the RPG programmer” on page 82
- ▶ 4.3, “Additional CL commands and useful ILE APIs” on page 139
- ▶ 4.4, “More information about ILE and shared open data paths” on page 140

4.1 Introduction to ILE

The integrated language environment (ILE) is a new programming model as compared to the original programming model (OPM). ILE was introduced with the IBM i operating system release V2R3 for the C language and with release V3R1 for the RPG IV language.

New concepts were introduced with the ILE programming model:

- ▶ New object types (modules, service programs, and binding directories).
- ▶ A substructure of the job structure called an *activation group*. An activation group is created when an ILE program or a service program is started or activated.
- ▶ New CL commands that enable collecting (binding) the objects together to work in activation groups.

The foundation of ILE and how it relates to RPG are explained in 4.2, “ILE tips for the RPG programmer” on page 82. Modules, service programs, binding directories, activation groups, and the CL commands that pull them all together are explained in the following sub-sections.

The following topics are covered in this section:

- ▶ 4.1.1, “Modules and binding” on page 78
- ▶ 4.1.2, “Service programs” on page 79
- ▶ 4.1.3, “Export and import” on page 79
- ▶ 4.1.4, “Binder language source” on page 79
- ▶ 4.1.5, “Binding directories” on page 80
- ▶ 4.1.6, “Activation groups” on page 80
- ▶ 4.1.7, “CL commands used with ILE and RPG” on page 81

4.1.1 Modules and binding

Modules are objects of *MODULE type that are created by the compiler when the Create RPG Module (**CRTRPGMOD**) command is used. A module can be composed of a main procedure (also referred to as main program) or one or more subprocedures.

Procedure: The term *procedure* often designates a subprocedure or a main procedure.

A module is sometimes called a *compilation unit* since it comes from the compilation of one source member. Modules are not executable, they only serve as building blocks for program creation. The process of program creation is called *binding*. Bound programs are executable objects of *PGM type.

To bind modules into a program, the Create Program (**CRTPGM**) command is used. If an RPG IV program does not call subprocedures or external modules, the Create Bound RPG Program (**CRTBNDRPG**) command works for both compilation and binding. This is the case, for example, of an RPG IV program converted from an RPG III program by the Convert RPG Source (**CVTRPGSRC**) command.

4.1.2 Service programs

If you have procedures that are called by more than one program, you could bind them individually to each of the programs. In such a case, they would occupy space in each program and would be difficult to maintain. If you group the procedures in a service program instead, the procedures occur only once and can be easily maintained.

Service programs are objects of *SRVPGM type, which are created by the Create Service Program (**CRTSRVPGM**) command. A service program is simply a collection of modules, especially those containing subprocedures. Service programs cannot be directly called. However, the procedures contained in them can be called by ILE programs or other procedures.

Service programs are built by binding, much like programs. But, they need to be further bound to a program before they are used. This is done by using the **CRTPGM** command.

Service programs can also be bound to other service programs. The top service program in such a group is eventually bound to a program using the **CRTPGM** command.

4.1.3 Export and import

A service program makes its own modules and procedures available to external users through a mechanism called *export*. The external users are modules and subprocedures in external programs and other service programs that use (call) the modules and subprocedures of the service program to call them. The external users are also called public or clients.

A service program exports its own subprocedures by specifying the EXPORT keyword in the subprocedure definition. However, to bring this specification in effect, the binding command (**CRTSRVPGM**) specifies which of the exported procedures are actually made available to external users.

Besides modules and subprocedures, variables can be exported by specifying the EXPORT keyword. Exported modules, subprocedures, and variables are collectively called *exports*. Exports are used in other procedures to which they are referred. The references are also called *imports* as opposed to the exports, which are sometimes called *definitions*.

4.1.4 Binder language source

The Create Service Program (**CRTSRVPGM**) command specifies how the service program is bound and what procedures and variables it exports. The EXPORT parameter of the command specifies how the exports are made available to the external users:

The EXPORT(*SRCFILE) value is the default and requires that a special source member exists. The source member, called *binder language source*, contains a list of exported subprocedure names (and possibly variable names) that the service program actually makes available. The other exports of the service program (except for module names) remain inaccessible to external users. The source member has a BND source type and is placed in a source file (the default is QSRVSRC). The binder source member is never compiled.

The EXPORT(*ALL) value makes available all exports from the service program.

Tip: Generally it is considered a best practice that you do not use EXPORT(*ALL) because it makes maintenance difficult.

4.1.5 Binding directories

Binding directories are objects of *BNDDIR type. Binding directories can be used as an additional source of exports. A binding directory contains a list of modules and service programs that are candidates for automatic binding.

Not all items contained in the list in the binding directory are necessarily bound. Only those required by imports that cannot otherwise be resolved are bound. Modules and service programs listed in a binding directory often contain standard procedures, for example, mathematical functions or other system procedures. Programmers can create their own binding directories using special CL commands.

4.1.6 Activation groups

Activation groups are temporary storage structures placed inside jobs (which themselves are also temporary structures). There are three types of activation groups: default, named, and new.

Default activation groups exist automatically and are never deleted. There are two default activation groups. Many system programs run in default activation group 1. RPG IV programs created with the parameter DFTACTGRP(*YES) of the **CRTBNDRPG** command run in default activation group 2.

Note: In this chapter, the name *default activation group* is used to mean activation group 2.

The other types of activation groups are specified by the parameter ACTGRP in the program and service program creation commands **CRTPGM** and **CRTSRVPGM**. Therefore, the type of activation group is determined by the program or service program at creation time.

An activation group is actually created when the program is started. An activation group can include:

- ▶ Static and automatic variables

The variables of programs running in the activation group. Static variables are those defined in a main procedure. They come from external sources such as DDS or SQL specifications, or they are defined as RPG variables (fields, indicators). One more place you will find static variables is as local variables in subprocedures declared with the STATIC keyword. Automatic variables are local variables defined in subprocedures.

- ▶ Open data paths (ODP)

Temporary objects representing open files to programs. The data buffer and pointer to the current record are part of the ODP.

- ▶ Dynamically allocated storage

Temporary object created by the %alloc BIF in the RPG IV program.

- ▶ Error handling routines

System or user programs (modules) handling error messages. Programmers can write their own modules to handle error messages coming from any procedure in the call stack, no matter in which programming language the procedure is written. Notice that the “program stack” has been renamed to “call stack”. For more information, go to “Call stack and error handling” on page 121“.

4.1.7 CL commands used with ILE and RPG

There are four basic commands that are used to create ILE modules, programs, and service programs:

- ▶ Create RPG Module (**CRTRPGMOD**) command
Invokes the ILE RPG compiler, which produces the *MODULE object.
- ▶ Create Program (**CRTPGM**) command
Invokes the ILE binder (independent of the programming language) and creates the *PGM program object from specified modules.
- ▶ Create Service Program (**CRTSRVPGM**) command
Invokes the ILE binder (also independent of the programming language) and creates the *SRVPGM object from specified modules.
- ▶ Create Bound RPG Program (**CRTBNDRPG**) command
Creates a module object in the QTEMP library and creates a *PGM program object from the module. The module is lost after the job ends.

Other commands enable change, display or delete functions. Useful display commands include:

- ▶ Display Module (**DSPMOD**) command
Displays module information on the screen or printer.
- ▶ Display Service Program (**DSPSRVPGM**) command
Displays service program information on the screen or a printer.
- ▶ Display Program (**DSPPGM**) command
Displays program information on the screen or a printer.

The following commands help to create binding directories that can be referred to by the **CRTSRVPGM** command as a source of suitable exports:

- ▶ Create Binding Directory (**CRTBNDDIR**) command
Creates a *BNDDIR object to be specified in the BNDDIR parameter of the **CRTSRVPGM** command.
- ▶ Add Binding Directory Entry (**ADDBNDDIRE**) command
Adds entries (module or service program names) to the binding directory.
- ▶ Work with Binding Directory Entries (**WRKBNDDIRE**) command
Displays binding directory entries (module or service program names) on the screen so you can maintain them.

Other commands service binding directories and their entries (delete and display commands).

4.2 ILE tips for the RPG programmer

This section illustrates the following ILE concepts when used with RPG IV:

- ▶ Various methods to create ILE programs from modules and service programs.
- ▶ Export and import of external symbols.
- ▶ Service program signatures and their relation to programs.
- ▶ Activation group creation and deletion, as well as shared open data paths and overrides.
- ▶ ILE specific error message handling program.

4.2.1 Creating programs from modules (binding by copy)

In this section short examples are presented to illustrate various methods of program creation. The sample programs are intentionally simple so you can concentrate on the mechanics of binding rather than on program logic.

The examples in this section use prototypes and procedure interfaces for all the calls. If you want to see what other calls you can use, see the *Programming IBM Rational Development Studio for i ILE RPG Programmer's Guide*:

https://www.ibm.com/support/knowledgecenter/ssw_ibm_i_73/rzasc/sc092507.pdf

A main procedure calling a subprocedure as a function

In this example, two modules, M02 and M02A, are created from two source members and bound into a program P02.

Module M02

Module M02 is compiled from the source shown in Example 4-1. Module M02 calls the subprocedure NonDigit in the IF statement as a function that returns a value. The value is *OFF if no non-digit character in the string is found or *ON if one exists. The position number of the first non-digit character is available in the second parameter.

Example 4-1 Module M02 source

```

//*****
// Member M02 from ILESRC
//
// Module M02 - function call to module M02A
//
//*****
ctl-opt main(mainPgm);

dcl-pr mainPgm extpgm('P02');
end-Pr;

// Data definitions
Dcl-s NonDigTxt char(30)      Inz('Non-digit in position');
Dcl-s AllDigits char(30)       Inz('All digits in string');

// Prototype for procedure NonDigit
/COPY ILESRC,CPYM02A

// Main procedure

```

```

dcl-proc mainPgm;
  dcl-pi mainPgm;
  end-Pi;

  // Local variables
  Dcl-s String      varchar(100)  Inz('111*1');
  Dcl-s Position    packed(3 : 0)  Inz(1);

  If NonDigit(String : Position);
    Dsply (NonDigTxt + ' ' + %char(Position));
  Else;
    Dsply AllDigits;
  EndIf;

  *InLR = *On;
end-Proc;

```

Module M02A

Module M02A does not contain a main procedure so it specifies the NOMAIN keyword. It contains only the subprocedure NonDigit. Module M02A is compiled from the source shown in Example 4-2.

Example 4-2 Module M02A source

```

//*****
// Member M02A from ILESRC
// Module M02A - function NonDigit
//*****
ctl-opt nomain;

// Procedure prototype
/COPY ILESRC,CPYM02A

// Procedure definition

dcl-Proc NonDigit EXPORT;
  // Procedure interface (must match the prototype)
  Dcl-pi NonDigit ind;
    String  varchar(100) value;
    Position packed(3 : 0);
  end-Pi;

  //Dump;
  Position = %check('0123456789' : String : 1);
  if position = 0;
    return *off;
  else;
    return *on;
  endIf;
end-Proc NonDigit;

```

Note: The EXPORT keyword which must be specified to make the procedure NonDigit accessible to the module (main procedure) M02. Note also that the function returns not only the return value (*ON or *OFF) but also passes the Position parameter by reference. Therefore, the calling module M02 can use it.

Prototype CPYM02A

The prototype CPYM02A is used as a copy member by both modules M02 and M02A. See Example 4-3.

Example 4-3 Prototype CPYM02A source

```
//*****
//  
//  CPYM02A from ILESRC  
//  
//  Prototype for procedure NonDigit  
//  
//*****  
  
Dcl-pr NonDigit ind;  
  String  varchar(100) value;  
  Position packed(3 : 0);  
end-Pr;
```

Try it yourself: Compilation of the two modules M02 and M02A can be done by using the following two commands:

```
CRTRPGMOD MODULE(RPGISCOOL/M02) SRCFILE(RPGISCOOL/ILESRC)  
CRTRPGMOD MODULE(RPGISCOOL/M02A) SRCFILE(RPGISCOOL/ILESRC)
```

Then, use the following command to bind the two modules together into the program P02:

```
CRTPGM PGM(RPGISCOOL/P02) MODULE(RPGISCOOL/M02 RPGISCOOL/M02A) ENTMOD(*FIRST)  
ACTGRP(QILE)
```

The ENTMOD parameter says that the first module, M02, gets control when the program P02 is started.

Use the following command to run the program:

```
CALL PGM(RPGISCOOL/P02)
```

A main procedure calling a nonfunctional subprocedure

In this example, two separate modules, M03 and M03A, are created from two source members and bound into a program P03.

Module M03

Module M03 is compiled from the source shown in Example 4-4 on page 85.

Module M03 calls the procedure NonDi that has the same purpose as the function NonDigit in the previous example. The call is now accomplished by a call with no return value. It provides the result of the call in the second parameter Position passed by reference. If the input string contains a non-digit character, the second parameter contains its position number (non-zero). Otherwise, it contains zero.

Example 4-4 Module M03 source

```

//*****
//
//  Member M03 from ILESRC
//
//  Module M03 - non-function call to a subprocedure in module M03A
//
//*****

ctl-opt main(mainPgm);

dcl-pr mainPgm extpgm('P03');
end-Pr;

//  Procedure prototype
Dcl-pr nonDi;
  string  varchar(100) Value;
  position packed(3 : 0);
end-Pr;

//  Data definitions
Dcl-s NonDigTxt char(30)      Inz('Non-digit in position ');
Dcl-s AllDigits char(30)       Inz('All digits in string');

//  Main procedure
dcl-proc mainPgm;

dcl-pi mainPgm;
end-Pi;

// Local variables
Dcl-s String    varchar(100)  Inz('111*1');
Dcl-s Position   packed(3 : 0) Inz(1);

//  Main procedure
NonDi  (String : Position);

If Position <> 0;
  Dsply (NonDigTxt + %char(Position));
Else;
  Dsply AllDigits;
EndIf;

return;

end-Proc mainPgm;

```

Module M03A

Module M03A does not contain a main procedure, so it specifies the NOMAIN keyword. It contains only the subprocedure NonDi. Module M03A is compiled from the source shown in Example 4-5.

Note: The EXPORT keyword must be specified to make the procedure NonDi accessible to the module (main procedure) M03.

Example 4-5 Module M03A source

```

//*****
// Member M03A from ILESRC in RPGISCOOL
//
// Module M03A - subprocedure - no function
//
//*****


ctl-opt Nomain;

// Procedure prototype
Dcl-pr nonDi;
    string varchar(100) Value;
    position packed(3 : 0);
end-Pr;

// Procedure start
dcl-Proc NonDi EXPORT;

// Procedure interface (must match the prototype)
Dcl-pi nonDi;
    string varchar(100) Value;
    position packed(3 : 0);
end-Pi;

Position = %check('0123456789' : String : 1);
return;
end-Proc NonDi;

```

Try it yourself: Compilation of the two modules can be done by using the following two commands:

```

CRTRPGMOD MODULE(RPGISCOOL/M03) SRCFILE(RPGISCOOL/ILESRC)
CRTRPGMOD MODULE(RPGISCOOL/M03A) SRCFILE(RPGISCOOL/ILESRC)

```

Then, use the following command to bind the two modules together into the program P03:

```

CRTPGM PGM(RPGISCOOL/P03) MODULE(RPGISCOOL/M03 RPGISCOOL/M03A)
ENTMOD(*FIRST) ACTGRP(QILE)

```

The ENTMOD parameter says that the first module, M03, gets control when the program P03 is started.

Use the following commands to run the program:

```
CALL PGM(RPGISCOOL/P03)
```

Passing data by EXPORT and IMPORT between two modules

Note: The method of passing data between modules by EXPORT and IMPORT is presented here only as a supplement to other methods. It is not usually needed. It could be used, for example, for passing data to a program that is called indirectly through an intermediate program that needs this data. However, such a requirement indicates poor program design.

In this example, two separate modules, M04 and M04A, are created from two source members and bound into a program P04.

Module M04

Module M04 is compiled from the source shown in Example 4-6.

This time, the first module, M04, does not use any parameters. It exports its two variables designated as EXPORT, instead, to make them accessible by the other module. The procedure call is used to call the module M04A. A prototype for M04A with no parameters is included in the code. Notice that EXPORT is specified in the module where the procedure is used.

Example 4-6 Module M04 source

```

//*****
// Member M04 from ILESRC
// Module M04 - CALLB to M04A bound module - export variables
//*****
// Data definitions

Dcl-s String      varchar(100)  Inz('111*1')  export;
Dcl-s Position    packed(3 : 0)   export;

Dcl-s NonDigTxt  char(30)       Inz('Non-digit in position');
Dcl-s AllDigits   char(30)       Inz('All digits in string');

dcl-pr M04A;
end-Pr;

// Main procedure
M04A();
If Position <> 0;
  Dsply (NonDigTxt + %char(Position));
Else;
  Dsply AllDigits;
EndIf;

return;

```

Module M04A

Module M04A is compiled from the source shown in Example 4-7.

The module M04A accepts two variables exported from the module M04 through the procedure call and processes them. Note that the variables (String and Position) are designated by the IMPORT keyword and have the same names in both modules.

This type of passing values between modules should not be used altogether. If still used, take care in maintenance because it represents “hidden parameters”. Such hidden passing can be also more difficult to debug than passing regular parameters.

Example 4-7 Module M04A source

```
//*****
// Member M04A from ILESRC
//
// Module M04A - main procedure - import variables
//
*****
```

```
// Data definitions
Dcl-s String      varchar(100)    import;
Dcl-s Position    packed(3 : 0)    import;

// Main procedure
Position = %check('0123456789' : String : 1);

return;
```

Try it yourself: Compilation of the two modules can be done by using the following two commands:

```
CRTRPGMOD MODULE(RPGISCOOL/M04) SRCFILE(RPGISCOOL/ILESRC)
CRTRPGMOD MODULE(RPGISCOOL/M04A) SRCFILE(RPGISCOOL/ILESRC)
```

Then, use the following command to bind the two modules together into the program P04:

```
CRTPGM PGM(RPGISCOOL/P04) MODULE(RPGISCOOL/M04 RPGISCOOL/M04A) ENTMOD(*FIRST)
ACTGRP(QILE)
```

The ENTMOD parameter says that the first module, M04, gets control when the program P04 is started.

Use the following command to run the program:

```
CALL PGM(RPGISCOOL/P04)
```

Creating service programs and binding by reference

The examples in this section show how modules and procedures are bound into service programs. Remember that service programs cannot be directly called. They need to be bound to a program. This kind of binding is called *bind by reference*. After the program is bound, it contains the service program name and the names of the program imports that correspond to names exported from the service program. These import names are called references and are resolved into pointers (addresses) at program activation time when the program is started and not earlier. The program does not contain a copy of the service program code.

In the following examples, there are two elementary procedures and a third procedure that uses them. These examples show how the three procedures (subprocedures) can be arranged in service programs and used in a program.

The first elementary procedure is DynEdit, which performs dynamic editing. It edits a 15-digit packed number defined with 0 decimal positions (15 0), as if it had a different number of decimal positions. This is sometimes needed in application packages where all numbers in database files are defined as (15 0) and the number of decimal positions is stored in separate numeric fields in another database file. No check is made if the number of decimal positions is greater than 15. The procedure is placed in module M11A.

The second elementary procedure is NonDigit, which checks if a string contains a valid number. The valid characters are digits and blanks. The procedure is placed in module M11B.

The third procedure is EdtChrNbr, which edits a character coded number, as if it had the requested decimal positions. This procedure uses both the elementary procedures in sequence. The input to the procedure is a string of digit and blank characters. If the string contains at least one invalid character (other than decimal digit or blank), the result is all blanks. The procedure is placed in module M11.

All three modules have the NOMAIN keyword. They are used (called) by module M10, which contains a main procedure and no subprocedures.

Source codes of the four modules used in the examples in this section are presented in “Source codes” on page 90. Then, different ways are shown how the modules can be combined in various service programs in the following sections:

- ▶ “Creating one service program” on page 98
- ▶ “Creating two chained service programs” on page 99
- ▶ “Creating three chained service programs” on page 100

The decision about grouping the modules into service programs (that is, how many service programs to create from these modules) is made as part of the application design. Service programs are simply collections of commonly used modules of code. In that sense, the kind of logic that is used for deciding how many IBM i libraries you need and what kinds of objects are grouped together by library is similar to the logic you will use here.

From a performance perspective, the groupings should be made based on how likely it is that the modules will be referenced together by a group of programs in the same job. Grouping multiple modules together into a single service program requires fewer connections between a program and service program to accomplish multiple tasks. Since making these connections takes time at application run time, minimizing the number of connections necessary can have a positive performance impact.

On the other hand, you can go too far in that direction. For example, you would not want to create only one extremely large service program for use by all programs in your entire shop. This is because the memory required to activate the large service program in every user job (if all the users are not using all the functions) can cause too much paging activity on the system. Likewise, the CPU time required to initialize all the storage in the service program modules that are not used by some users could cause a performance problem.

In addition to the performance impact of service program packaging, you should consider the impact on maintaining the applications. The packaging should be set up so that the programmers using the functions can easily find them to use and maintain them. Some sort of logic in the groupings, such as grouping similar functions together, is useful in this respect.

As you can see, the decisions about packaging modules into service programs is a balancing act. You must try to balance the performance needs with ease of development and maintenance. Performance needs also require a balance between the desire to minimize the number of connections between any given program and its required service programs. At the same time, you must minimize the activation and initialization of modules in a service program that are not used by a group of programs that a given user utilizes within a job.

The good news is that changing the service program packaging, if it turns out you made the wrong decisions in the beginning, is a relatively easy task. If the module objects still exist on the system, no re-compile of the source code is required to change the packaging scheme.

For this particular example of the DynEdit, NonDigit, and EdtCharNbr procedures, it is most likely that these modules would best be grouped together into a single service program (as in the first example). This is because all three modules are used together by at least one main module (M10). Grouping them reduces the number of connections required between program and service programs. In addition, all three procedures are similar in function. They all perform various types of string handling functions, so there is also a logical reason to group them together. As a matter of fact, you could even argue that it might have been a good idea to group all three of these subprocedures into a single module because of the close relationship they have to one another. However, for purposes of illustrating the technical possibility of grouping them in different ways, we have chosen to put the three functions into separate modules for this illustration.

How the service programs are bound to a program is explained in “Binding service programs to programs” on page 101.

Source codes

This section lists the following source codes of the four modules used in the examples in this section:

- ▶ “Procedure DynEdit in module M11A” on page 90
- ▶ “Procedure NonDigit in module M11B” on page 92
- ▶ “Prototype CPYS11” on page 93
- ▶ “Procedure EdtChrNbr in module M11” on page 94
- ▶ “Program (main procedure) in module M10” on page 95
- ▶ “Prototype CPYS10” on page 96
- ▶ “Display file description CHRNUMW” on page 97
- ▶ “Try it yourself: The following commands can be used to create the modules described above. These compilations needs to be done before creating the service program objects described in the upcoming sections.” on page 98

Procedure DynEdit in module M11A

Module M11A is compiled from the source shown in Example 4-8.

Example 4-8 Module M11A source

```
//*****
// Member M11A from ILESRC
//
// Module M11A - Dynamic editing
// Contains one subprocedure - DynEdit
//
*****
```

ctl-opt nomain;

```

//-----
//  Procedure prototypes
//-----
/Copy ILESRC,CpyS11

//-----
//  Procedure definition
//-----
dcl-Proc DynEdit EXPORT;

//-----
//  Procedure interface
//-----
Dcl-pi DynEdit varchar(100);
    Number    packed(15 : 0) Value;
    DecPos   packed(3 : 0)  Value;
end-Pi;

//-----
//  Local data definitions
//-----
Dcl-s EdtNbr      varchar(100);
Dcl-s WorkNbr     packed(30 : 15);
Dcl-s Correction  packed(1 : 0);
Dcl-s Pos         packed(3 : 0);

//  If requested decimal positions are 0 - the correction is -1

If DecPos = 0;
    Correction = -1;

//  If requested decimal positions are > 0 - the correction is 0
Else;
    Correction = 0;
EndIf;

//  Shift the (15 0) input number right by requested decimal places
//  (if decimal places are positive or zero) and place it to
//  (30 15) work variable. E.g. DecPos = 2:
//          1           1           2           3
//  1...5....0....5   1...5....0....5....0....0
//  1111111111111111 ==> 00111111111111110000000000000000

WorkNbr = Number * 10 ** -DecPos;

//  Edit the work number with edit code 3 and place the result
//  in the varying length character variable:
//  ' 1111111111111.1100000000000000'

EdtNbr = %EdtC(WorkNbr :'3');

//  Extract the edited number without trailing digits:
//  Text from position 1 to the end, minus 15,
//          plus dec.positions,

```

```

//      ' 1111111111111.11'           plus correction

EdtNbr = %Subst(EdtNbr : 1 :
                  %Len(EdtNbr) -15 + DecPos
                  + Correction );

//    Return the edited number as a varying character variable

Return EdtNbr;

end-proc DynEdit;

```

Procedure NonDigit in module M11B

Module M11B is compiled from the source shown in Example 4-9.

Example 4-9 Procedure M11B source

```

//*****
// Member M11B from ILESRC
//
// Module M11B - Test for nondigit character
//                 Contains one procedure - NonDigit
//
//*****

ctl-opt Nomain;

//=====
// NonDigit - Checks if the input character variable contains
//             a non-digit (or nonblank) character.
//             If yes, returns error code *On and replaces
//             the input variable with all zero characters.
//             If not, returns positive code *Off replaces
//             all blanks by zeros.
//
//=====

//-----
// Procedure prototypes
//-----

/Copy ILESRC,CpyS11

//-----
// Procedure definition
//-----

dcl-Proc NonDigit EXPORT;

//  Procedure interface (must match the prototype)
Dcl-pi NonDigit ind;
  String  varchar(100) value;
  Position packed(3 : 0);
end-Pi;

```

```

//-----  

//  No local data definitions  

//-----  

//  Check if invalid character is found in the string  

//      (other than a digit or blank)  

Position = %check(' 0123456789' : String : 1);  

//  If found replace the input string with all zero digits  

//  Return *On if invalid character found, *Off if not found  

if position <> 0;  

    String = *All'0';  

    return *on;  

else;  

    //  If all digits or blanks - Replace blanks by zeros  

    String = %xlate(' ' : '0' : String);  

    return *off;  

endif;  

end-Proc NonDigit;

```

Prototype CPYS11

These prototypes are used by the copy member in modules M11, M11A, and M11B as shown in Example 4-10.

Example 4-10 Prototype CPYS11 source

```

//*****  

//  

//  Member CPYS11 from ILESRC  

//  

//  Prototypes for functions: DynEdit - Dynamic editing  

//                                NonDigit - Check for non-digit  

//                                characters in a string  

//*****  

//  Prototype for function DynEdit - Dynamic editing  

Dcl-pr DynEdit varchar(100);  

    Number  packed(15 : 0) Value;  

    DecPos  packed(3 : 0) Value;  

end-Pr;  

//  Prototype for function NonDigit - Check for non-digit  

//                                characters in a string  

Dcl-pr NonDigit ind;  

    String  varchar(100) value;  

    Position packed(3 : 0);  

end-Pr;

```

Procedure EdtChrNbr in module M11

Module M11 is compiled from the source shown in Example 4-11.

Example 4-11 Procedure M11 source

```

//*****
// Member M11 from ILESRC
// Module M11 - Edit character coded number
// Contains one procedure - EdtChrNbr
//*****
ctl-opt Nomain;

//=====
// EdtChrNbr - Edit character coded number
//=====

//-----
// Procedure prototypes
//-----
/Copy ILESRC,CpyS10
/Copy ILESRC,CpyS11

//-----
// Procedure definition
//-----
dcl-Proc EdtChrNbr  EXPORT;

Dcl-pi EdtChrNbr ind;
    CharNbr  varchar(100);
    DecPos   packed(3 : 0) Value;
    EditedNbr varchar(100);
end-pi;

//-----
// Local data definitions
//-----

Dcl-s Pos   packed(3 : 0);
Dcl-s Number packed(15 : 0);

// Check if the characters contain all digits or blanks
If NonDigit(CharNbr : Pos);

    // If not all digits or blanks - Return *On (error)
    Return *On;
EndIf;

// Else (all digits or blanks) - Convert characters to a number
Number = %dec(CharNbr : 15 : 0);

// Edit the number with requested decimal positions (edit code 3)

```

```

EditedNbr = DynEdit(Number : DecPos);

//  Return *Off (OK)
Return *Off;

end-Proc EdtChrNbr;

```

Program (main procedure) in module M10

Module M10 is compiled from the source shown in Figure 4-12.

Example 4-12 Program M10 source

```

//*****
// Member M10 from ILESRC
//
// Module M10 - Entering character coded numbers and edit them
//               with requested decimal positions on the screen
//
//*****

ctl-opt main(mainPgm);

//=====
// File description - Display file
//=====
dcl-F CHRNUMW  WorkStn;

//=====
// Data definitions
//=====
Dcl-s EditNbr varchar(100);
Dcl-s RC      ind;
Dcl-s CharNbr varchar(100);

// Called procedure prototypes
/Copy ILESRC,CpyS10

dcl-pr mainPgm extpgm('S123');
end-Pr;

//=====
// Mainline program
//=====
dcl-proc mainPgm;

// Process display file
DoW 1 = 1;

// Show the first format to enter a character number
// and requested decimal positions to edit
ExFmt CHRNUMWR;
If *In03;
  Leave;
EndIf;

```

```

// Edit the character coded number with requested decimal positions
CharNbr = %trim(CHRNB);
RC = EdtChrNbr(CharNbr: DecPos: EditNbr);

// If error - Supply all blanks
If RC;
    EDTNBR = *Blanks;
Else;

    // Else (OK) - Trim trailing blanks and shift right
    EvalR EDTNBR = %TrimR(EditNbr);
EndIf;

// Show the result on the second screen format
ExFmt CHRNWMR2;
If *In03;
    Leave;
EndIf;

EndDo;                                // End do while forever

end-Proc;

```

Prototype CPYS10

Prototype CPYS10 is used by the copy member in module M10 and M11 as shown in Example 4-13.

Example 4-13 Prototype CPYS10 source

```

//*****
// Member CPYS10 from ILESRC
//
// Prototype for function EdtChrNbr - Edit character coded number
//
//*****

// EdtChrNbr prototype - edit a character into a number
Dcl-pr EdtChrNbr ind;
    CharNbr  varchar(100);
    DecPos   packed(3 : 0) Value;
    EditedNbr varchar(100);
end-pr;

```

Display file description CHRNUMW

The display file description shown in Example 4-14 is used in the main procedure of module M10. This is used to define the user interface screen shown in the following screen examples.

Example 4-14 DDS screen display file for M10 source

```
*****
* Member CHRNUMW from ILESRC
*
* CHRNUMW - Edit character coded numbers
*
*****
```

A DSPSIZ(24 80 *DS3)
A CA03(03 'End')

* Format to enter a character number and required decimal pos.
A R CHRNUMWR
A 5 4'Enter a number without special characters:'
A DSPATR(HI)
A CHRNBR 15A B 6 5
A 8 4'Enter number of decimal positions -
A you want to have in the edited number:
A DSPATR(HI)
A DECPOS 3 0B 9 5EDTCDE(4)

* Format to show resulting edited number
A R CHRNUMWR2
A 4 4'Character coded number:'
A CHRNBR 15A 0 6 5
A 8 4'Required decimal positions:'
A DECPOS 3Y 00 9 5EDTCDE(3)
A 12 4'Resulting edited number:'
A EDTNBR 16A 0 13 5DSPATR(RI)
A 16 5'Press Enter.'
A COLOR(BLU)
A 23 2'F3=Exit'
A COLOR(BLU)

Format CHRNUMWR2 in Figure 4-1 shows the result of editing along with the entered values.

The screenshot shows a terminal window titled 'A - ETCB4T1.RCHLAND.IBM.COM'. The window displays the following text:

```

Character coded number:
111111111111
Required decimal positions:
2

Resulting edited number:
1111111111.11

Press Enter.

F3=Exit

```

The bottom status bar shows 'MA A' and '01/001'.

Figure 4-1 Result of editing a character coded number

Try it yourself: The following commands can be used to create the modules described above. These compilations needs to be done before creating the service program objects described in the upcoming sections:

```

CRTRPGMOD MODULE(RPGISCOOL/M11A) SRCFILE(RPGISCOOL/ILESRC) SRCMBR(*MODULE)
CRTRPGMOD MODULE(RPGISCOOL/M11B) SRCFILE(RPGISCOOL/ILESRC) SRCMBR(*MODULE)
CRTRPGMOD MODULE(RPGISCOOL/M11) SRCFILE(RPGISCOOL/ILESRC) SRCMBR(*MODULE)

```

Prior to creating the module M10, the display file CHRNUMW must have been created by using the following commands:

```

CRTDSPF FILE(RPGISCOOL/CHRNUMW) SRCFILE(RPGISCOOL/ILESRC)
ADDLIB LIB(RPGISCOOL)
CRTRPGMOD MODULE(RPGISCOOL/M10) SRCFILE(RPGISCOOL/ILESRC) SRCMBR(*MODULE)

```

Creating one service program

You can bind all three procedures into one service program by using the following **CRTSRVPGM** command:

```

CRTSRVPGM SRVPGM(RADREDBOOK/S123) MODULE(RADREDBOOK/M11A RADREDBOOK/M11B
RADREDBOOK/M11) EXPORT(*ALL) ACTGRP(*CALLER)

```

Figure 4-2 on page 99 illustrates how the service program is built combining the three separate modules into one service program.

The service program is called S123 and made all its exports available to external callers. The **EXPORT(*ALL)** parameter on the **CRTSRVPGM** command specifies exactly this.

Note: The exports are the names of procedures DynEdit, NonDigit, and EdtCharNbr, which specify the EXPORT keyword. You will see later that this option, although easy to specify, hides in itself some disadvantages related to signatures. A signature is a code that expresses a version of exports. It is similar to level check values used with files.

The ACTGRP(*CALLER) parameter on the **CRTSRVPGM** command indicates that the service program runs in the same activation group as its caller. The caller is the program to which the service program is bound. Several different programs can bind the same service program and call its procedures.

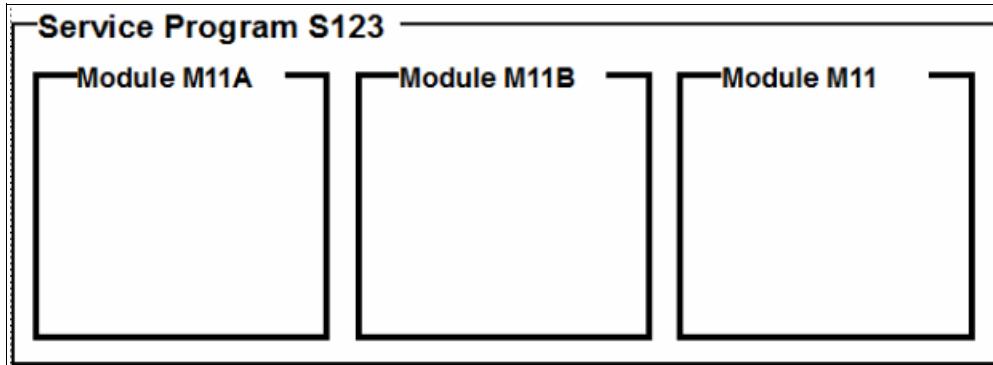


Figure 4-2 Service program bound from three modules

Creating two chained service programs

You can create two separate service programs and bind them together. First, service program S12 is created by binding modules M11A and M11B (DynEdit and NonDigit procedures). Service program S3S12 is then created by binding module M11 and the service program S12. The service program S12 is chained to the service program S3S12. The corresponding commands are:

```
CRTSRVPGM SRVPGM(RADREDBOOK/S12) MODULE(RADREDBOOK/M11A RADREDBOOK/M11B)
EXPORT(*ALL) ACTGRP(*CALLER)
```

```
CRTSRVPGM SRVPGM(RADREDBOOK/S3S12) MODULE(RADREDBOOK/M11) EXPORT(*ALL)
BNDSRVPGM(RADREDBOOK/S12) ACTGRP(*CALLER)
```

Figure 4-3 on page 100 illustrates how the two separate service programs are chained together to create one entity for the program to resolve.

The two service programs are called S12 and S3S12 and they “export all its exports” by using the EXPORT(*ALL) parameter on the **CRTSRVPGM** command.

The imports from the module M11 (procedures DynEdit and NonDigit) were resolved in the example in “Creating one service program” on page 98 by specifying Bind Service Program (BNDSRVPGM) S12. A different possible way to resolve them would have been to include modules M11A and M11B in the same service program as in the previous example with service program S123.

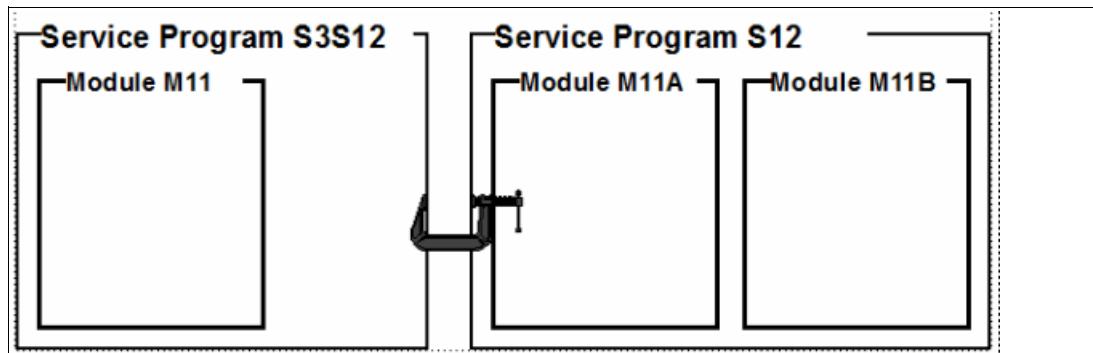


Figure 4-3 Two chained service programs

Creating three chained service programs

Service program S3S1S2 is created if you bind module M11 with two service programs, S1 and S2. The following commands show how to do it:

```
CRTSRVPGM SRVPGM(RADREDBOOK/S1) MODULE(RADREDBOOK/M11A) EXPORT(*ALL)
```

```
CRTSRVPGM SRVPGM(RADREDBOOK/S2) MODULE(RADREDBOOK/M11B) EXPORT(*ALL)
```

```
CRTSRVPGM SRVPGM(RADREDBOOK/S3S1S2) MODULE(RADREDBOOK/M11) EXPORT(*ALL)
BNDSRVPGM(RADREDBOOK/S1 RADREDBOOK/S2) ACTGRP(*CALLER)
```

Figure 4-4 illustrates how the three separate service programs are chained together to form a single entity for the program to resolve.

Note: A service program always binds at least one module and an arbitrary number of service programs.

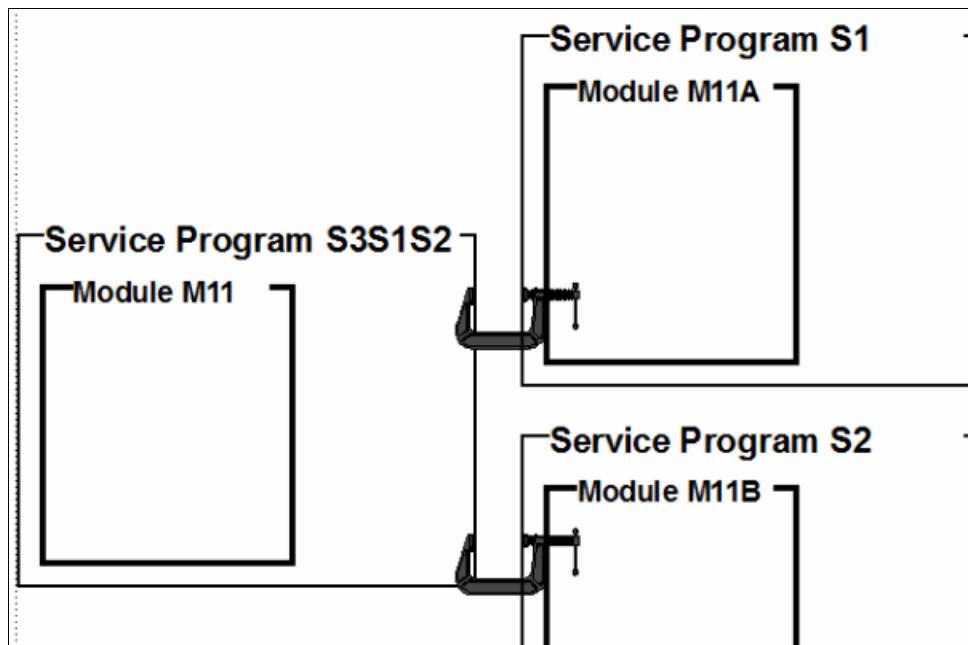


Figure 4-4 Three chained service programs

4.2.2 Binding service programs to programs

Now that you have several arrangements of service programs, you can create several programs, each binding a different set of service programs but still each performing the same function. Note that only three combinations are shown in this section:

- ▶ “Program from one module and one service program” on page 101
- ▶ “Program from one module and a chained service program” on page 102
- ▶ “Program from two modules and two service programs” on page 102

Important: Recall that programs are objects of *PGM type and can be run. Modules are objects of *MODULE type and cannot be run. Service programs are objects of *SRVPGM type and cannot be run without a program.

Program from one module and one service program

Program PS123 is created by binding module M10 (main procedure) by copy and the service program S123 (by reference). Figure 4-5 illustrates how the program is built by chaining with the services program.

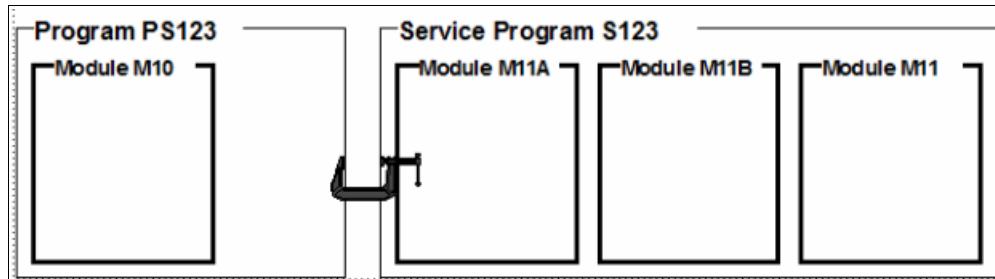


Figure 4-5 Program consisting of one module and one service program

Try it yourself: Use the following command to create the program object using the service program created in “Creating one service program” on page 98:

```
CRTPGM PGM(RPGISCOOL/PS123) MODULE(RPGISCOOL/M10) ENTMOD(*FIRST)
BNDSRVPGM(S123) ACTGRP(QILE)
```

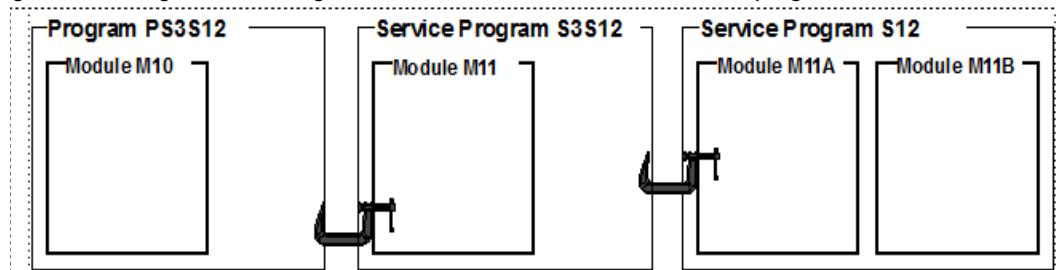
Run program PS123 using the following commands:

```
ADDLIB LIB(RPGISCOOL)
CALL PGM(RPGISCOOL/PS123)
```

Program from one module and a chained service program

Program PS3S12 is created by binding module M10 and the chained service program S3S12. Figure 4-6 illustrates how the program is built chaining multiple service programs together.

Figure 4-6 Program consisting of one module and two chained service programs



Try it yourself: Use the following command to create the program object using the service programs created in “Creating two chained service programs” on page 99:

```
CRTPGM PGM(RPGISCOOL/PS3S12) MODULE(RPGISCOOL/M10) ENTMOD(*FIRST)
BNDSRVPGM(RPGISCOOL/S3S12) ACTGRP(QILE)
```

Run program PS3S12 by using the following commands:

```
ADDLIBL LIB(RPGISCOOL)
CALL PGM(RPGISCOOL/PS3S12)
```

Program from two modules and two service programs

Program PM11S1S2 is created by binding modules M10 and M11 (by copy) and service programs S1 and S2 in parallel (by reference). Figure 4-7 illustrates how the program consisting of two modules is chained with two service programs.

Note: A program always binds at least one module. It does not need to bind any service program at all, or it can bind several service programs.

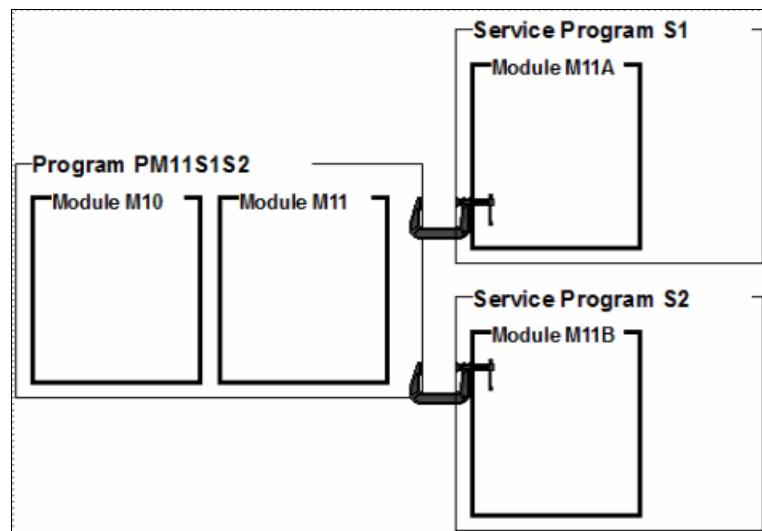


Figure 4-7 Program consisting of two modules and two service programs

Try it yourself: Use the following command to create the program object using the service programs created in “Creating three chained service programs” on page 100 on page 80:

```
CRTPGM PGM(RPGISCOOL/PM11S1S2) MODULE(RPGISCOOL/M10 RPGISCOOL/M11)  
ENTMOD(*FIRST) BNDSRVPGM(RPGISCOOL/S1 RPGISCOOL/S2) ACTGRP(QILE)
```

Run program PM11S1S2 by using the following commands:

```
ADDLIB LIB(RPGISCOOL)  
CALL PGM(RPGISCOOL/PM11S1S2)
```

4.2.3 Service programs, binder language, and signatures

When you created a service program, you used the EXPORT(*ALL) parameter on the **CRTSRVPGM** command. This allows all exports of all modules in the service program to be used by other applications.

All public exports form a base for the signature, which is a value that is derived by the binder as a unique characteristic of the service program.

The signature of a service program has a similar function as the level check value used in files. Whenever a service program is activated by the program, a check is made to see if the signature matches the one that is stored in the program since the last binding. If not, an error message occurs. To correct this error situation, the developer needs to rebind the service program to all programs that reference it. Note that this is only a rebind requirement and not a recompile requirement. The easiest and most common way to accomplish this rebind is by using the Update Program (**UPDPGM**) command and specifying the service program that has changed. This way, you do not need to have all the modules at the correct version to create the program again using the original **CRTPGM** command.

Another method to use is the parameter EXPORT(*SRCFILE) along with two other parameters designating a source member containing binder language source. The Source Entry Utility (SEU) type for the binder language source member is BND and its default source file is QSRVSRC. The specifications in this binder language source are collectively called *binder language*.

Some customers prefer maintaining binder language to manage the service program signatures to rebinding all the programs that use a service program. This is especially true in cases where the only change made to the service program was to add one or two new procedures. If you have only a single IBM i system running your application, you might find that rebinding using the **UPDPGM** command is simpler than creating and maintaining the binder language source. In fact, many ILE programmers go through their entire careers without every using binder language. However, if you have many production systems, particularly if they are in remote locations, you might find that maintaining binder language makes it easier to make relatively small, incremental changes to your service program structures. If there was a need to rebind all the programs using the changed service program, you would need to either perform that rebind on all the remote systems or you would need to rebind on your system and ship all the updated program objects and all the changed service program objects to the remote locations.

In addition, by using the binder source, you can control which procedures from the service program you actually want to make available to calling programs by controlling which ones you export. This gives you the flexibility to “hide” some of the procedures in the service program so that they are callable only from other procedures inside the same service program. Then, they will not be callable by program modules or other service program modules.

Binder language

Binder language consists of three statements and comments. Comments look much the same as those in CL language. The statements are:

- ▶ **STRPGMEXP**: Starts a block of export symbols and has three parameters.
- ▶ **EXPORT**: Specifies an external (export) symbol, for example, a subprocedure or variable name.
- ▶ **ENDPGMEXP**: Ends the block of export symbols.

Example 4-15 shows two blocks of export symbols. In the first block, default values for the parameters in the **STRPGMEXP** statement are shown. If you do not specify any parameters, these are assumed.

Example 4-15 Binder language source example

```
/* Current version of exports */
    STRPGMEXP PGMLVL(*CURRENT) LVLCHK(*YES) SIGNATURE(*GEN)
        EXPORT      SYMBOL(s1) /* translates to S1 */
        EXPORT      SYMBOL('s2') /* remains as s2 */
        EXPORT      SYMBOL("s3") /* remains as s3 */
    ENDPGMEXP

/* Previous version of exports */
    STRPGMEXP PGMLVL(*PRV)
        EXPORT      SYMBOL(s1)
        EXPORT      SYMBOL('s2')
    ENDPGMEXP
```

Following are the parameters of the **STRPGMEXP** statement:

- ▶ Parameter **PGMLVL** can have the following values:
 - *CURRENT: The currently valid block of export symbols. Only one block of this type is allowed and should be the first block.
 - *PRV: The block represents a previous version of the export symbols. There can be more than one previous block.
- ▶ Parameter **LVLCHK** can have the following values:
 - *YES: The signature is checked by the system at activation time.
 - *NO: The signature is not checked.

Note: It is considered a best practice that you do not use the **LVLCHK** option because it can cause unpredictable results. Other options, such as specifying a signature value, allow for the necessary flexibility.

- ▶ Parameter **SIGNATURE** can have the following values:
 - *GEN: The signature code is generated by the binder.

- value: The signature code is supplied by the programmer as a 16-byte character value expressed in text or hexadecimal notation. This option is discussed “Using your own signatures” on page 106.

The EXPORT statement has only the SYMBOL parameter, which specifies a procedure name or a variable name that is to be exported from the service program. The name is sometimes called “export symbol” or “external symbol”. The name can be written in capital and small letters without apostrophes or quotation marks. In this case, the name is converted into capital letters. If lower case letters or special characters are needed in exported symbols, apostrophes or quotes need to be used.

If a new export symbol is to be added to the list of exports, the current block can be expanded by adding the new export symbol after the last existing EXPORT statement.

Using blocks of exported symbols

The service program S12 could and should use the following binder source specified in member S12 of the QSRVSRCS source file in library RADREDBOOK:

```
STRPGMEXP PGMLVL(*CURRENT)
  EXPORT SYMBOL("DYNEDIT")
  EXPORT SYMBOL("NONDIGIT")
ENDPGMEXP
```

This is the simplest form of the binder language source. The source member is usually named the same as the service program it belongs to as in the S12 example.

The binder language source is used in the CRTSRVPGM command as follows:

```
CRTSRVPGM SRVPGM(RADREDBOOK/S12) MODULE(RADREDBOOK/M11) EXPORT(*SRCFILE)
SRCFILE(RADREDBOOK/QSRVSRCS) SRCMBR(*SRVPGM) ACTGRP(*CALLER)
```

The EXPORT, SRCFILE, and SRCMBR parameters are needed to specify the binder language source. The *SRVPGM value in the SRCMBR parameter tells the binder that the source member has the same name as the resulting service program. You can specify your own name instead. However, using the service program name is considered a good practice.

The Retrieve Binder Source (**RTVBNDSRC**) command can be used to help generate the binder language source based on exports from one or more modules. For example, the following command generates the source text in member S12 of the ILESRC file as shown in Example 4-16.

```
RTVBNDSRC MODULE(RADREDBOOK/M11A RADREDBOOK/M11B) SRCFILE(RADREDBOOK/ILESRC)
SRCMBR(S12)
```

Example 4-16 S12 binder source

```
STRPGMEXP PGMLVL(*CURRENT)
/*****
/*  *MODULE      M11A          RADREDBOOK  01/22/16 11:38:36   */
/*****
  EXPORT SYMBOL("DYNEDIT")
/*****
/*  *MODULE      M11B          RADREDBOOK  01/22/16 11:38:36   */
/*****
  EXPORT SYMBOL("NONDIGIT")
ENDPGMEXP
```

Each block of export symbols defined by the EXPORT statements between **STRPGMEXP** and **ENDPGMEXP** represents a separate signature. The sequence in which the symbols are ordered is significant for the signature (if it is generated by the binder). Each change in the sequence causes a change in the signature.

The block of currently valid exports is specified by the PGMLVL(*CURRENT) parameter of the **STRPGMEXP** statement. One or more blocks of previously valid exports can be specified with the PGMLVL(*PRV) parameter.

The binder stores all signatures in the service program so that they are available when a program is bound and later called. The current signature is stored in the program when it is created (bound) by the **CRTPGM** command.

At program activation time, the system checks if the signature stored in the program matches one of those in the service program. If at least one signature matches, the program can run. If there is no matching signature, a level check error is reported (if not disabled by LVLCHK(*NO) in the binder language source). This way, the system ensures that the correct version of the service program is used with the program. This is similar to level checking with files.

The “previous” signatures might help you to run new versions of service programs with old versions of a program without rebinding. The service program “remembers” the previous signatures. This allows a program that has stored the old (current when the program was compiled) service program signature to use the newly re-compiled service program.

Using your own signatures

There are occasional circumstances where you might find it helpful or necessary to hard code your own signature values using binder language. When doing this, be careful to ensure that the sequence of the previously used exports remains exactly the same as it was when any programs were bound to it. Any change in the sequence of exports or any removal of exports from earlier versions of this service program causes seriously negative effects when programs connect to the recreated service program. It is quite possible that if the list of exports is not maintained correctly, an incorrect procedure could be called and run by mistake. If the system generates your signatures for you (the Signature *GEN option), it creates signatures based on the list in the binder language. It is certainly possible for you to write the binder language incorrectly so that they are specified in the wrong sequence. Likewise, it is even more possible that exports could get into the wrong sequence if you do not specify them in the binder language at all.

You might want to consider explicitly specifying a signature for special circumstances. For example, you might want to intentionally force an incompatible signature for a service program because you made significant changes to a particular procedure's function and need to ensure that all programs that use that service program are reviewed and updated as necessary to use the new function appropriately. Forcing a change to the service program signature ensures that any programs that were not updated would get an error when called. This is preferable to using the function in the service program incorrectly.

Your signature specification might appear as either or the following possibilities:

```
STRPGMEXP PGMLVL(*CURRENT) SIGNATURE('AnIncompatibleSi')
STRPGMEXP PGMLVL(*CURRENT) SIGNATURE(X'000000000000000F105631521336215')
```

You can determine the signature values in a service program by running the following display Service Program (**DSPSRVPGM**) command:

```
DSPSRVPGM SRVPGM(RADREDBOOK/S12) OUTPUT(*PRINT) DETAIL(*SIGNATURE)
```

The printout looks similar to what is shown in Example 4-17.

Example 4-17 Results of the DSPSRVPGM command

```
*....+....1....+....2....+....3....+....4....+....5....+....6....+....7....+
                                         Display Service Program Information
5770SS1 V7R2M0 140418
  Service program . . . . . : S12
  Library . . . . . : RADREDBOOK
  Owner . . . . . : DIEPHUIS
  Service program attribute . . . . . : RPGLE
  Detail . . . . . : *SIGNATURE
                                         Signatures:
000000000000000F105631521336215
```

For more information about signatures, refer to Programming ILE Concepts, SC41-5606:

https://www.ibm.com/support/knowledgecenter/ssw_ibm_i_73/ilec/sc415606.pdf

4.2.4 Using binding directories

Binding directories are created by two CL commands:

- ▶ Create Binding Directory (**CRTBNDDIR**)
- ▶ Add Binding Directory Entry (**ADDBNDDIRE**) or Work with Binding Directory Entries (**WRKBNDDIRE**)

The **CRTBNDDIR** command creates an object of *BNDDIR type. The **ADDBNDDIRE** command adds a new entry to the directory list. Each entry contains an object name. The objects are modules or service programs.

Binding directory QC2LE: If you work with functions that are defined for the C language, you specify the system binding directory QC2LE in your **CRTRPGMOD** or **CRTBNDRPG** command. You can specify it on the H specification in your source program as the keyword BNDDIR('QC2LE'). This binding directory lists service program names that export the function names.

The program PM11S1S2 is created by using binding directory BM11S1S2. The binding directory BM11S1S2 is created by the following commands:

```
CRTBNDDIR BNDDIR(RADREDBOOK/BM11S1S2)
ADDBNDDIRE BNDDIR(RADREDBOOK/BM11S1S2) OBJ((RADREDBOOK/M11 *MODULE))
ADDBNDDIRE BNDDIR(RADREDBOOK/BM11S1S2) OBJ((RADREDBOOK/S1 *SRVPGM))
ADDBNDDIRE BNDDIR(RADREDBOOK/BM11S1S2) OBJ((RADREDBOOK/S2 *SRVPGM))
```

You can display contents of a binding directory by using the **DSPBNDDIR** or **WRKBNDDIRE** command:

```
DSPBNDDIR BNDDIR(RADREDBOOK/BM11S1S2) OUTPUT(*PRINT)
```

The printout looks similar to what is shown in Example 4-18.

Example 4-18 Results of the DSPBNDDIR command

*....+....1....+....2....+....3....+....4....+....5....+....6....+....7....+... Display Binding Directory					
5770SS1 V7R2MO 140418 Binding Directory : BM11S1S2				Library . -----Creation-	
Object	Type	Library	Activation	Date	Ti
M11	*MODULE	RADREDBOOK		01/22/16	11
S1	*SRVPGM	RADREDBOOK	*IMMED	01/07/16	14
S2	*SRVPGM	RADREDBOOK	*IMMED	01/07/16	14

Now you can create your program by the binder command:

```
CRTPGM PGM(RADREDBOOK/PM11S1S2) MODULE (RADREDBOOK/M10) BNDDIR(RADREDBOOK/BM11S1S2)  
ACTGRP(QILE)
```

This creates the same binding as the earlier example in “Program from two modules and two service programs” on page 102. The program still performs the same function as the previous programs.

Binding directories are used in the operating system to bind system procedures to compiled modules. The names of system procedures are generated by the compiler as unresolved imports (references). For example, you can display your module contents by the command:

```
DSPMOD MODULE (RADREDBOOK/M11B) DETAIL(*IMPORT)
```

Then, you would get the following system procedure names:

```
_QRNX_SUBP_EXCP  
_QRNX_XLATE_1  
_QRNX_INIT_H  
_QRNX_INIT  
_QRNX_GET_ASSOC_CCSID  
_QRNX_SIGNAL_EXCP  
Q LE 1eDefaultEh2
```

These unresolved imports are later resolved through system binding directories. The binder can find them in the service program QRNXIE listed in the binding directory QRNXLE in the QSYS library.

You can combine modules and service programs in many variations to produce a program. The specific combination you choose depends on your preferences or your software project conventions.

4.2.5 Activation groups

Activation groups enable programs to run in isolated environments. Every ILE program and service program contains information about the activation group in which it runs. The **CRTPGM** and **CRTSRVPGM** binder commands have a parameter, ACTGRP, which specifies the activation group. Also the Create Bound RPG Program (**CRTBNDRPG**) command accepts the ACTGRP parameter if you specify DFTACTGRP(*NO).

Defining and creating activation groups

The Create Program (**CRTPGM**) and Create Bound RPG Program (**CRTBNDRPG**) commands allow the following values for the ACTGRP parameter:

- ▶ *NEW: A new activation group is created every time the program is called. Using this value for all ILE programs can result in serious performance problems. The *NEW option allows recursive calling of RPG IV programs because a new set of static variables and open data paths are created each time the program is called. However, creation of a new activation group is a resource and time consuming process. You should use this option only if you actually need it.
- ▶ *ENTMOD: The value causes the program entry procedure (PEP) module to be examined:
 - If the module attribute is RPGLE, CBLLE, or CLLE then either the QILE or QILETS activation group is used.
 - If the storage model is single level, QILE is used.
 - If the storage model is terospace, QILETS is used.
 - If the module attribute is not one of the above, *NEW is used. This is the default value for the **CRTPGM** command.
- ▶ Name: A named activation group is created if the program is called and the activation group does not yet exist. This is the most preferable option if there is only one program per application and everything else is a service program as it enables separation of applications in a most efficient way. When there are multiple programs in an application, it should be used on the main program that is calling all the others. In this case, all the other programs should use *CALLER or exactly the same name.
- ▶ *CALLER: No new activation group is created. The program runs in the existing activation group from which the program was called. It is the ideal solution when an application is made of multiple programs and the main one has been invoked in a NAMED activation group.

The Create Service Program (**CRTSRVPGM**) command allows either *CALLER or name for activation groups:

- ▶ Name: A named activation group is created if the service program procedure is called and the activation group does not already exist.
- ▶ *CALLER: No new activation group is created. The service program procedures run in the activation group from which they were called. This is the default value. This is the most preferred solution if they are called from different programs at the same time. Then, they have separate resources for each individual call.

The Create Bound RPG Program (**CRTBNDRPG**) command has one other option if the default activation group parameter is *NO (DFTACTGRP):

- ▶ *STGMDL:
 - If the storage model parameter (STGMDL) is single level (*SNGLVL), the program runs in the QILE activation group.
 - If the storage model parameter is terospace (TERASPACE), the program runs in the QILETS activation group.
 - If the storage model parameter is inherit (*INHERIT) the activation group parameter must be *CALLER and the program runs in the activation group of the calling program.

A note about storage models

Sometimes a program or service program needs to allocate something larger than 16 megabytes. This is doable if the terospace storage model is used. The default storage model is single level storage, which limits any variable or object to 16 megabytes. The terospace storage model allows around 100 terabytes of storage for a program.

An activation group can be single level or terospace. The first program called in an activation group determines the storage model. The default activation group, where OPM programs run, can only be single level. If an activation group is started as terospace you can not use any program or service program that was created as single level storage. If an activation group is single level you can not call any program or procedure that was created as terospace.

Programs and service programs can be created to inherit the storage model of the activation group, but only if activation group is specified as *CALLER when they are created. Any ILE program, that needs to use terospace, that is called by an OPM program must be created to run in a different activation group (DFTACTGRP(*NO)).

Ending subprocedures, programs, and activation groups

It is important to know how subprocedures, programs (main procedures), and activation groups can be ended or deleted:

- ▶ A subprocedure is ended by the RETURN operation or by reaching its last statement. Then all local (automatic) variables are removed from the stack, except the static variables (those with the STATIC keyword) are retained.
- ▶ A program (main procedure) is ended by:
 - Issuing the RETURN operation without turning on the LR indicator. In this case, all files remain open (open data paths are still available) and static variables remain untouched.
 - Setting the LR indicator on and then returning. In this case, the files are closed (open data paths are deleted), and static variables remain allocated but are marked for future initialization. If the main procedure is called later again, the static storage is initialized without a new allocation.
- ▶ An activation group is ended (deleted) according to its type:
 - A *NEW activation group is completely deleted when the program ends abnormally or when it returns to its caller (even without the LR indicator turned on). This is important when calling a program recursively and also when you test a new program. You do not need to sign off or use a Reclaim Activation Group (**RCLACTGRP**) command to end the activation group as you would have to with a named activation group.
 - A named activation group exists during the life of the active job. It is ended along with the job end. To end a named activation group while the job is still running, you can use the **RCLACTGRP** command or a bound call to the ILE API program, **CEETREC**. The **CEETREC** call has no required parameters. While the **CEETREC** call ends the current activation group immediately, the **RCLACTGRP** command can specify an activation group name or *ELIGIBLE. The eligible activation groups are those where no programs or procedures are currently in the call stack for the job. Only activation groups that are not in use can be deleted with this command. A named activation group is deleted automatically if the only program active in it ends abnormally.
 - The Default activation group is never ended before the job ends.

Overriding files in activation groups

If you want to change some characteristics of a file (database, printer, display, and so on), you can use an **OVRxxxF** command, where xxx stands in place of DB for database or PRT for printer, and so on. For example, you can use the Override Data Base File (**OVRDBF**) command to override the SHARE parameter of a database file, or the Override with Printer File (**OVRPRTF**) command to override the form length or the printer device name.

The override commands have, among others, two parameters that specify a scope (boundary) of their influence. The OVRSCOPE parameter tells the system where in the call stack the override will be valid, before the file is open. The OPNSCOPE parameter specifies where in the call stack the first (full) open operation on the file will be valid for subsequent (shared) open operations on the same file.

The OVRSCOPE parameter can specify the following values:

- ▶ *ACTGRPFDN: The scope of the override is determined by the activation group of the program that issues 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 whole activation group of the calling program.
- ▶ *CALLLVL: 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 lower (numerically higher) than the current call level are influenced by this override.
- ▶ *JOB: The scope of the override is the job in which the override occurs.

The OPNSCOPE parameter can specify the following values:

- ▶ *ACTGRPFDN: The scope of the open operation is determined by the activation group of the program that called the **OVRxxxF** 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.
- ▶ *JOB: The scope of the open operation is the job in which the open operation occurs.

The Open Data Base File (**OPNDBF**) command has the OPNSCOPE parameter, which can have similar values, so you can specify it at open time instead of overriding in advance:

- ▶ *ACTGRPFDN: The scope of the open operation is determined by the activation group of the program that called the **OPNDBF** 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.
- ▶ *ACTGRP: The scope of the open data path (ODP) is the activation group. Only those 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 Close File (**CLOF**) command closes the activation group.
- ▶ *JOB: The scope of the open operation is the job in which the open operation occurs.

Sharing an open data path

An open data path (ODP) is the path through which all input and output operations for a file are performed. Usually a separate open data path is defined each time a file is opened. If you specify SHARE(*YES) for the file permanently or temporarily, the first program's open data path for the file is shared by subsequent programs that open the same file. You specify SHARE(*YES) permanently with the **CRTxxxF** or **CHGxxxF** commands. Temporary specification is done by using the override (**OVRxxxF**) commands.

The point is that for a given shared ODP, a single file cursor (record pointer) exists. Therefore, file operations in separate programs that use that shared ODP will affect each other.

Let's say a program positions to a specific record (for example, with SETLL) in a file with a shared ODP and calls a second program, which performs an I/O operation (for example, READ, CHAIN, or SETLL). The first program is no longer positioned at that record. It is positioned to whatever record the second program selected. For example, if the first program reads record 6, it can expect that the next sequential read returns record 7. However, if a second program sharing the ODP is called and it chains to record 11, the first program reads record 12.

If a program holds a record lock in a file with a shared ODP and then calls a second program which performs an I/O operation (for example, READ, READ with no lock, UPDATE, DELETE, or UNLOCK) on the same file, the first program no longer retains the record lock.

For ILE programs running in non-default activation groups, shared files are scoped to either the job level or the activation group level. Shared files that are scoped to the job level can be shared by any programs running in any activation group within the job. Shared files that are scoped to the activation group level can be shared only by the programs running in the same activation group.

For programs running in non-default activation groups, the default scope for shared files is the activation group. For job-level scope, specify OPNSCOPE(*JOB) on the override command (often the **OVRDBF** command).

The RPG IV language offers several enhancements in the area of shared open data paths. If a program or procedure performs a read operation, another program or procedure can update the record as long as SHARE(*YES) is specified for the file in question. In addition, when using multiple-device files, if one program acquires a device, any other program sharing the ODP can also use the acquired device. It is up to the programmer to ensure that all data required to perform the update is available to the called program.

Sharing an open data path improves performance because the system does not have to create a new open data path. However, sharing an open data path can cause problems. For example, an error is signaled in the following cases:

- ▶ If a program sharing an open data path attempts file operations other than those specified by the first open (for example, attempting input operations although the first open specified only output operations).
- ▶ If a program sharing an open data path for an externally described file tries to use a record format that the first program ignored.
- ▶ If a program sharing an open data path for a program described file specifies a record length that exceeds the length established by the first open.

Shared open data path in a common activation group

Although shared open data paths save processing time for open operations, they can sometimes lead to unwanted results, as the example in Figure 4-8 shows.

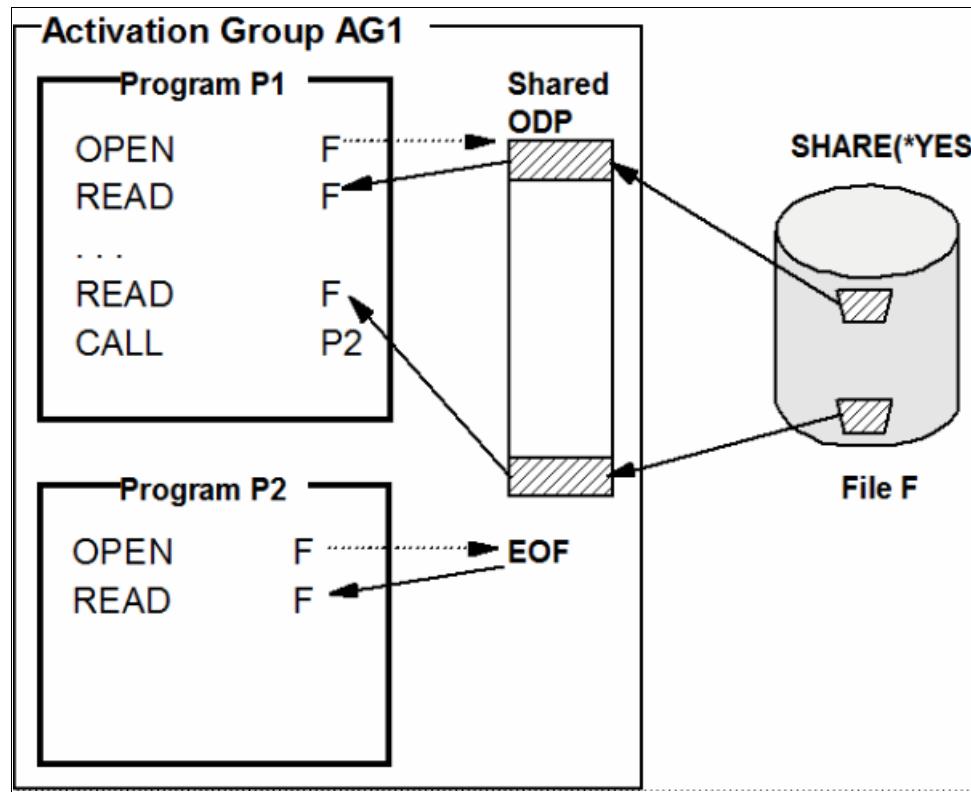


Figure 4-8 Shared open data path in a common activation group

The following series of events occurs:

1. Program P1 opens file F (creates the open data path) and reads all records sequentially.
The current record position is at the end of file (EOF).
2. Program P1 calls the program P2.
3. Program P2 opens file F by using the same ODP with the current record position at the end of file (EOF).
4. Program P2 tries to read a record but none is available.

Of course, if the program P2 issued a SETLL operation immediately after opening the file, it looks better. If the program P2 comes from a different vendor than program P1, it might be difficult to achieve.

This situation is more specifically presented in the following examples. Programs P1 and P2 print the contents of the ITEMS database file, which specifies SHARE(*YES) permanently. In addition, program P2 updates the file by increasing the unit price by one percent. Both programs open the file for update because they share the same file. If program P1 specified only input, an error would occur when opening the file in program P2.

Program P1

Program P1 is compiled from the source code that is shown in Example 4-19.

Example 4-19 Program P1 source

```

//*****
// Member P1  from ILESRC
//
// Program P1 - Sequential processing of a file
//
//*****ctl-opt DFTACTGRP(*NO) ACTGRP('AG1') main(mainPgm);      1

// File descriptions

dcl-F ITEMS Disk(*ext) usage(*update);
dcl-F REPORT printer;

// prototypes
dcl-pr mainPgm extpgm('P1');
end-Pr;

dcl-pr P2 extpgm('P2');
end-Pr;

// Main procedure
dcl-proc mainPgm;
  dcl-pi  mainPgm;
end-Pi;

Read(N) ITEMS;

If %EoF;                                         2
  EOFTEXT = 'P1 End of file';
Else;
  EOFTEXT = 'P1 Beginning of file';
EndIf;
Write EOFLINE;

Dow Not %EoF;                                     3
  Write ITEMDETAIL;
  Read(N) ITEMS;
EndDo;

P2();                                                 4

*inlr = *on;
Return;

end-Proc;

```

Note the following points, which correspond to the numbers shown in the right side of Example 4-19:

1. Program P1 runs in activation group AG1. It is created by the **CRTBNDRPG** command.

2. After the first READ operation, the program tests whether the current record position is EOF and, according to the test result, prints a line with appropriate text.
3. If the position is not at EOF, the program reads all database file records and prints them.
4. After it reads all records, the program calls the other program, which is P2.

Program P2

Program P2 is compiled from the source code that is shown in Example 4-20.

Example 4-20 Program P2 source

```

//*****
// Member P2  from ILESRC
//
// Program P2 - Sequential processing of a file
//
//*****ctl-opt DFTACTGRP(*NO) ACTGRP('AG1')  main(p2);      1

// File descriptions
dcl-F ITEMS  Disk(*ext) usage(*update);
dcl-F REPORT printer;

// prototypes
dcl-pr P2 extpgm('P2');
end-Pr;

// Main procedure
dcl-proc P2;
dcl-pi P2;
end-Pi;

Read ITEMS;

If %EoF;                                         2
  EOFTEXT = 'P2 End of file';
Else;
  EOFTEXT = 'P2 Beginning of file';
EndIf;
Write EOFLINE;

DoW Not %EoF;                                     3

  UNITPR = UNITPR * 1,01;
  Update ITEMSP;

  Write ITEMDETAIL;

  Read ITEMS;
EndDo;

*InLR = *On;
Return;
end-Proc;

```

Note the following points, which correspond to the numbers shown in the right side of Example 4-20 on page 115:

1. Program P2 runs in the same activation group, AG1, as program P1.
2. The record position is still at EOF because the database file is shared, which is true even if program P1 ended with LR indicator *ON. The program prints an end of file message.
3. The reading cycle is skipped, and program P2 ends and returns to the program P1, which also ends.

The programs must be compiled by running the **CRTBNDRPG** command because they specify the keywords **DFTACTGRP** and **ACTGRP** in the control spec. These keywords are not allowed in the **CRTRPGMOD** command because the **ACTGRP** parameter must be specified with the subsequent **CRTPGM** command.

File ITEMS

This example uses a physical file member with the definition that is shown in Example 4-21.

Example 4-21 Physical file source for database description

```
*****
*      Member ITEMS from ILESRC
*      ITEMS - Item master file
*              A sample database physical file
*
*****
```

A		UNIQUE	
A	R ITEMSP		
*	Item number		
A	ITEMNBR	5	COLHDG('Item' 'number')
*	Unit price		
A	UNITPR	9 2	COLHDG('Unit' 'price')
*	Item description		
A	ITEMDESC	50	COLHDG('Item description')
*	Key field		
A	K ITEMNBR		

The contents of the sample physical file that are used in these examples is shown in Example 4-22.

Example 4-22 Sample physical file contents

Item number	Unit price	Item description
00001	26.78	First item
00002	53.59	Second item
00003	80.38	Third item

Printer file REPORT

The printer file definition that is used in these sample programs is shown in Example 4-23.

Example 4-23 Printer file support

```
*****
*   Member REPORT from ILESRC
*
*   REPORT - List of items
*
*****
```

A		REF(ITEMS)
A	R ITEMDETAIL	SPACEA(1)
A	ITEMNBR R	2
A	UNITPR R	+2 EDTCDE(Q)
A	ITEMDESC R	+2
A	R EOFLINE	SPACEA(1)
A	EOFTEXT	50 2

Try it yourself: After running the example by calling program P1 from the command line, you get the following printout from program P1:

```
P1 Beginning of file
00001      25.00 First item
00002      50.00 Second item
00003      75.00 Third item
```

Create the programs P1 and P2 by using the following commands. Before you create the programs, the physical file and printer file must be created:

```
ADDLIBL RPGISCOOL
CRTPF FILE(RPGISCOOL/ITEMS) SRCFILE(RPGISCOOL/ILESRC)
CRTPRTF FILE(RPGISCOOL/REPORTS) SRCFILE(RPGISCOOL/ILESRC)
CRTBNDRPG PGM(RPGISCOOL/P1) SRCFILE(RPGISCOOL/ILESRC)
CRTBNDRPG PGM(RPGISCOOL/P2) SRCFILE(RPGISCOOL/ILESRC)
```

To run the program, run the following command:

```
CALL PGM(RPGISCOOL/P1)
```

The contents of the sample physical file that used in the examples is shown in Example 4-24.

Example 4-24 P1 sample file contents

P1 Beginning of file
00001 25.00 First item
00002 50.00 Second item
00003 75.00 Third item

You also get the following printout from program P2:

P2 End of file

The activation group AG1 remains active because it is a named activation group.

Instead of specifying SHARE(*YES) on the ITEMS file permanently, you can call the program P1 from a CL program. It specifies the override for SHARE(*YES) and is compiled as an OPM CL program of source type CLP by the Create CL Program (**CRTCLPGM**) command. The CL program contains the following commands:

```
OVRDBF      FILE(ITEMS) OVRSCOPE(*ACTGRPDFN) SHARE(*YES) OPNSCOPE(*ACTGRPDFN)
CALL        PGM(P1)
```

The **OVRDBF** command is issued when the CL program is called in the default activation group. In this case, *ACTGRPDFN is the same as *CALLLVL. It means that the open data path is shared between programs P1 and P2, and the results are the same as before.

If you change the source type to CLLE (ILE CL), you can compile the same CL program with the Create Bound CL Program (**CRTBNDCL**) command. Specify the activation group ACTGRP(AG1) for the CL program. In this case, the **OVRDBF** command causes the parameter SHARE(*YES) to be valid for open operations inside the activation group AG1 only. The results are again the same because programs P1 and P2 are running in the same activation group AG1.

Open data paths in different activation groups

If the programs run in different activation groups, each program creates and uses its own open data path. The paths cannot be shared across activation group boundaries when the SHARE(*YES) parameter is specified. The situation is shown in Figure 4-9. If another program that is using the ITEMS file was called in activation group AG1, it can share the ODP that is created by program P1A.

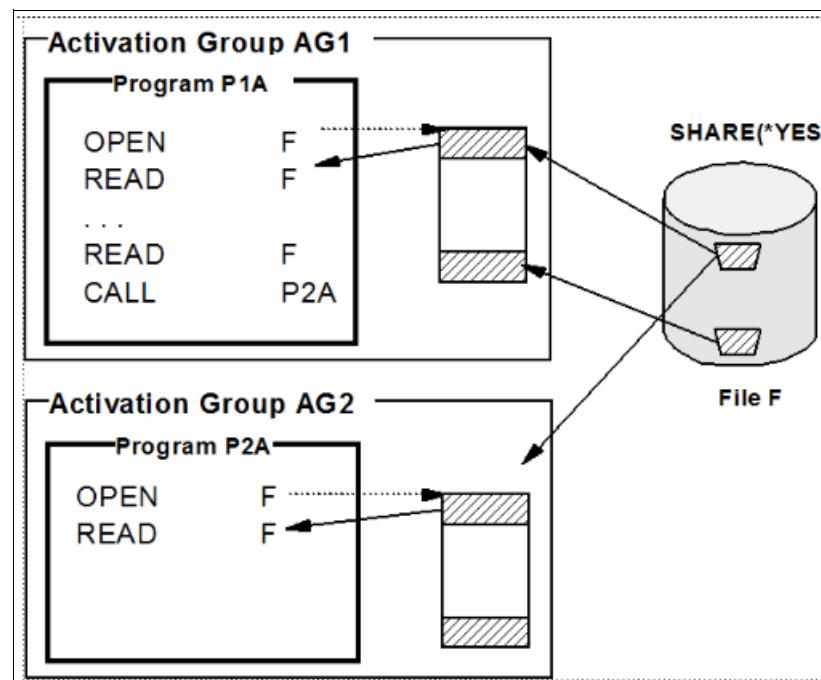


Figure 4-9 Open data paths in different activation groups

Note these points:

- ▶ Program P1A runs in activation group AG1 and ends its processing at EOF, which is the same for program P1 in "Program P1" on page 114.
- ▶ Program P2A runs in activation group AG2. It opens its own data path and reads records from the beginning of the file, no matter where the program P1A stopped reading.

This situation is specifically presented in the following examples. Program P1A and P2A perform the same function as programs P1 and P2.

Program P1A

Program P1A is identical to program P1, except that it calls program P2A and prints slightly different text, as shown in Example 4-25.

Example 4-25 Program P1A source

```

//*****
// Member P1A  from ILESRC
//
// Program P1A - Sequential processing of a file
//
//*****ctl-opt DFTACTGRP(*NO) ACTGRP('AG1') main(mainPgm);

// File descriptions

dcl-F ITEMS Disk(*ext) usage(*update);
dcl-F REPORT printer;

// prototypes
dcl-pr mainPgm extpgm('P1A');
end-Pr;

dcl-pr P2A extpgm('P2A');
end-Pr;

// Main procedure
dcl-proc mainPgm;
dcl-pi  mainPgm;
end-Pi;

Read(N) ITEMS;

If %EoF;
  EOFTEXT = 'P1A End of file';
Else;
  EOFTEXT = 'P1A Beginning of file';
EndIf;
Write EOFLINE;

Dow Not %EoF;
  Write ITEMDETAIL;
  Read(N) ITEMS;
EndDo;

P2A();

*inlr = *on;
Return;
end-Proc;

```

Program P2A

Program P2A is identical to P2, except it specifies activation group AG2 in its H specification and prints slightly different text. Example 4-26 shows its source code.

Example 4-26 Program P2A source

```

//*****
// Member P2A  from ILESRC
//
// Program P2A - Sequential processing of a file
//
//*****
ctl-opt DFTACTGRP(*NO) ACTGRP('AG2') main(p2A);

// File descriptions
dcl-F ITEMS Disk(*ext) usage(*update);
dcl-F REPORT printer;

// prototypes
dcl-pr P2A extpgm('P2A');
end-Pr;

// Main procedure
dcl-proc P2A;
  dcl-pi P2A;
  end-Pi;

  Read ITEMS;

  If %EoF;
    EOFTEXT = 'P2A End of file';
  Else;
    EOFTEXT = 'P2A Beginning of file';
  EndIf;
  Write EOFLINE;

  DoW Not %EoF;

  UNITPR = UNITPR * 1,01;
  Update ITEMSPR;

  Write ITEMDETAIL;

  Read ITEMS;
EndDo;

*InLR = *On;
Return;
end-Proc;

```

After running the example by calling program P1A from the command line, you get the following printout from program P1A:

```
P1A Beginning of file
00001      25.00 First item
00002      50.00 Second item
00003      75.00 Third item
```

You also get the following printout from program P2A:

```
P2A Beginning of file
00001      25.25 First item
00002      50.50 Second item
00003      75.75 Third item
```

Program P2A opens its own data path in the newly created activation group AG2 regardless of whether program P1A already has an ODP. Even though the file specifies SHARE(*YES) permanently, no sharing occurs between activation groups AG1 and AG2 because the first open was performed in the activation group AG1. Therefore, all records are printed in both programs, with the unit price increased by 1% in program P2A. The activation groups AG1 and AG2 remain active because they are named activation groups.

Using OVRSCOPE(*CALLLVL) does not cause the ODP to be shared.

If you want to share the ODP in both activation groups, run the following commands, with the open scope being the entire job:

```
OVRDDBF    FILE(ITEMS) OVRSCOPE(*JOB) SHARE(*YES) OPNSCOPE(*JOB)
CALL       PGM(P1A)
```

The **OVRSCOPE** parameter says that sharing the ITEMS file lasts for the entire time that the job is active (unless another override comes in between). The **OPNSCOPE** parameter says that subsequent opens of the ITEMS file after the first open follow the attributes that are set by this command. The attributes are, among others, open options for all types of access (input, output, update, and delete) and the share option set by the override command just before.

Try it yourself: You can create the programs P1A and P2A by running the following commands. Before you create the programs, the physical file and printer file must be created (if they were not created in the previous section):

```
ADDLIBLE RPGISCOOL
CRTPF FILE(RPGISCOOL/ITEMS) SRCFILE(RPGISCOOL/ILESRC)
CRTPRTF FILE(RPGISCOOL/REPORTS) SRCFILE(RPGISCOOL/ILESRC)
CRTBNDRPG PGM(RPGISCOOL/P1A) SRCFILE(RPGISCOOL/ILESRC)
CRTBNDRPG PGM(RPGISCOOL/P2A) SRCFILE(RPGISCOOL/ILESRC)
```

To run the program, run the following command:

```
CALL PGM(RPGISCOOL/P1A)
```

4.2.6 Call stack and error handling

The program stack from OPM has been renamed *call stack* because programs and subprocedures are contained in it.

Call stack and control boundaries

Certain entries in the call stack are known as *control boundaries*. An entry is a control boundary if it was created by calling an OPM program or a procedure (program or subprocedure) from a different activation group from the one before it. Often, a main procedure represents a control boundary, especially if called dynamically, that is, by the **CALL** statement from an ILE procedure or an OPM program.

The main procedure of an ILE program has a special name in the call stack, `_QRNP_PEP_programname`.

Program entry procedure (PEP) represents the main procedure of a program. The program name itself follows the PEP in the call stack. The PEP is a piece of code that is generated by the compiler to initialize (activate) the program. The PEP passes control to the program code that was written by the programmer.

Service programs and OPM programs have no PEP. Subprocedures and modules that are called by the bound call also have no PEP.

Entries appear in the call stack and disappear from it, as they are called and ended (in last in, first out (LIFO) order).

As an example, the call stack is shown in Figure 4-10 in two views.

Display Call Stack							
System: ETCB4T1							
Job:	QPADEV0004	User:	DIEPHUIS	Number:	094002		
Thread:	000000C2						
Type	Program	Statement	Procedure				
	QCMD	QSYS	/04FA				
1	QCMD	QSYS	/01C8				
	E01REG	RADREDBOOK	_QRNP_PEP_E01REG				
	E01REG	RADREDBOOK	150	E01REG			
	QRNXIO	QSYS	63	_QRNX_WS_EXFMT			
	QWSGET	QSYS	/0663				
	QT3REQIO	QSYS	/0253				
Display Call Stack							
System: ETCB4T1							
Job:	QPADEV0004	User:	DIEPHUIS	Number:	094002		
Thread:	000000C2						
----- Activation Group ----- Control							
Type	Program	Name	Number	Boundary			
	QCMD	QSYS	*DFTACTGRP	0000000000000001 Yes			
1	QCMD	QSYS	*DFTACTGRP	0000000000000001 No			
	E01REG	RADREDBOOK	QILE	0000000000000014 Yes			
	E01REG	RADREDBOOK	QILE	0000000000000014 No			
	QRNXIO	QSYS	QILE	0000000000000014 No			
	QWSGET	QSYS	*DFTACTGRP	0000000000000001 No			
	QT3REQIO	QSYS	*DFTACTGRP	0000000000000001 No			

Figure 4-10 Call stack for program E01REG

Program E01REG is waiting in the EXFMT operation. The top screen shows the entry _QRNP_PEP_E01REG. The second screen show us that QRNPPEP_E01REG is the next higher control boundary. If the < sign precedes the name, a shortened name of a system procedure is displayed. To see the full name, place the cursor in the name and press F22.

Error handling in ILE

If an error message arises in a procedure, it is processed by various program functions in the following order:

1. The (E) modifier in some operations, which can be tested by the %ERROR built-in function handles the error message.
2. An ILE condition handler, which can handle escape, status, and notify error messages coming from any procedure.
3. The INFSR subroutine (file information subroutine), which handles the error messages resulting from file I/O operations. The INFSR keyword cannot be specified if the file will be accessed by a subprocedure, or if NOMAIN is specified on the control option specification.
4. The *PSSR subroutine (program status subroutine), which is defined in a subprocedure, and each subprocedure can have its own *PSSR. The *PSSR in a subprocedure is local to that subprocedure. If you want the subprocedures to share the same exception routine, you should have each *PSSR call a shared procedure.
5. The default RPG exception handler.

The ILE condition handler is a an ILE function, and the other functions are also available in OPM.

For example, if a CHAIN (E) operation was specified and the record to be read is locked by another job, you would catch the error message by using the %ERROR built-in function.

If you specify a CHAIN operation without the (E) modifier, you can catch the error message by specifying an INFSR subroutine in the main procedure.

If you did not specify any of the preceding functions, you can catch error messages by using a condition handler program. You write it as a module with a special interface. You register the module by calling a special ILE API (CEEHDLR) in the program or the procedure in which you expect errors. The condition handler receives control whenever an error message arrives in the program or procedure message queue. You can test for messages that have type *STATUS, *NOTIFY, *ESCAPE, and function check, which is a special type of an *ESCAPE message (CPF9999).

The RPG specific functions (1, 3, and 4 from the above list), when applied, mark the message as handled, and the message is not propagated any further.

The last function, the default RPG exception handler, does not mark the message, but sends it to the message queue of the next higher entry in the stack. There, the message is processed again according to the hierarchy described above. This process is called *percolation* and continues until the message is handled or, a control boundary is reached.

If the message is not handled by the procedure in the control boundary, the system handles the *ESCAPE message as follows:

1. The *ESCAPE message is written in the job log, and a function check message (CPF9999) is generated and written in the job log.
2. The system returns to the point where the *ESCAPE message was originated and repeats the percolating process with the function check message.

3. At the control boundary, a RNQxxxx inquiry message can be generated and sent to the *EXT message queue if the control boundary is just below the command line. This only applies to main procedures.
4. The CEE9901 application error *ESCAPE message is generated and sent to the message queue of the call stack entry just above the control boundary.
5. For *STATUS and *NOTIFY error message types, see *IBM i ILE Concepts*, SC41-5606:
https://www.ibm.com/support/knowledgecenter/ssw_ibm_i_73/ilec/sc415606.pdf

ILE condition handler interface

ILE conditions are OS/400 exception messages that are represented in a manner that is independent of the system. An ILE condition token is issued to represent an ILE condition. Condition handling refers to the ILE functions that allow you to handle errors separately from language-specific error handling. You can use condition handling to handle error messages in programs composed of modules written in different ILE languages. 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 condition handler program accepts input parameters (as described in Table 4-1) that tells the kind of message it received and from which procedure.

Table 4-1 Input parameters accepted by the condition handler program

Variable Name	Description	RPG IV Data Type	Values
CondToken	Incoming message identification	Char(12)	See Table 4-2 on page 125.
ProcName	Name of the procedure that generated the message	Char(10)	Reference to the name (a pointer).
RtnAct	Return action: What the system should do with the message	Integer(10)	See the following Note.
NewToken	A new token value	Char(12)	New message information in case of promotion.

Note: RtnAct is a result code that the condition handler sends to the system to take a specific action. The values for the result code are as follows:

10 Resume at the next instruction, 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.

20 Percolate to the next condition handler.

21 Percolate to the next call stack entry. This can skip a high-level language condition handler for this call stack entry.

30 Promote to the next condition handler.

31 Promote to the next call stack entry. This may skip a high-level language condition handler for this call stack entry.

Condition token is a 12-byte data structure that contains information identifying the incoming message. Figure 4-11 describes the layout of the structure.

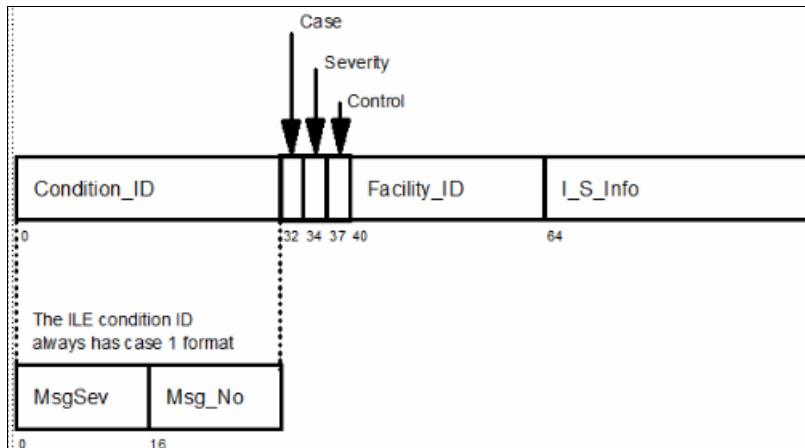


Figure 4-11 ILE condition token layout

The description of the specific contents of the structure is shown in 4-2 and applies to NewToken.

Table 4-2 Condition token structure

Variable name	Description	RPG IV data type	Values
MsgSev	Message severity	Integer(5)	0, 1, 2, 3, or 4.
MsgNo	Message number	Integer(5) or Char(2)	For example, X'1211' to divide by zero.
CaseSevCtl	Case, Severity, Control in one byte	Char(1)	See the following Note.
MsgType or FacilityID	Message type	Char(3)	For example, CEE, CPF, or MCH RNX.
MsgKey or ISInfo	Unique message key	Char(4)	A unique binary code.

Note: Case, Severity, and Control are binary numbers that are placed in bit fields in a byte:

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 Severity and MsgSev fields contain the same information. Actual used numbers are 0 - 4.
Control	A 3-bit field containing flags that describe or control various aspects of condition handling. The third bit specifies whether the Facility_ID has been assigned by IBM.

The message type and message number uniquely identify the incoming message.

You can process a message so that you replace it with a different message and send it to the system. This way of handling messages is called *promotion*. The result code is 30 or 31 in this case. Promotion is accomplished using the ILE API program CEENCOD.

The CEENCOD API program builds a new message according to the parameters that are listed in Table 4-3.

Table 4-3 Parameters for the CEENCOD API program to promote a message

Variable name	Description	RPG IV data type	Values
MsgSeverity	Message severity	Integer(5)	0 to 4
MsgNumber	Message number	Integer(5) or Char(2)	For example, X'1211'
Case	Format of message	Integer(5)	Always 1
Severity	Message severity	Integer(5)	Same as MsgSeverity
Control	A control flag	Integer(5)	0, 1
FacilityID	Message type	Char(3)	Three letters, for example, USR (see note)
MsgKey	Unique message key	Char(4)	A unique binary code
NewToken	Structured as the condition token	Char(12)	New message information
A variable name or *OMIT	Feedback information from CEENCOD	Char(12) or omitted	*OMIT or a 12-byte token

Note: Message file QUSRMSG must be created by running the Create Message File (**CRTMSGF**) command to enable promotion of USRxxxx messages. Message descriptions that are prefixed with USR must be added in the message file by running the **ADDMMSGD** command. You can change the message file name to QxxxMSG, but the messages must begin with xxx because you cannot specify the message file name in the parameters.

With this information, the CEENCOD API program builds a new condition token in the NewToken variable, which is sent (promoted) to the caller as a new message by the condition handler.

An example of the condition handler

A condition handling program is illustrated under the name ILEERRHDL in Example 4-27. The data definition part of the program is shown first in Example 4-27.

Example 4-27 Program ILEERRHDL source

```

//*****
// Member ILEERRHDL from ILESRC
//
// ILEERRHDL - ILE error handling program
// The program is called as soon as an error message comes that
//     is not handled by means of RPG IV error indicator or %Error
//     built in function, or by an INFSR subroutine.
// The program must be registered in a procedure.
//
//*****
ctl-opt nomain;

//=====
// Data definitions
//=====

// Program prototypes                               1
/COPY ILESRC,ILEERRPR

// Message information structure (CondToken and NewToken)      2
Dcl-s CondTokenPtr pointer;

Dcl-ds CondTokenDS based(CondTokenPtr);
  MsgSev      int(5);
  MsgNum      char(2);
  CaseSevCtl  char(1);
  MsgType     char(3);
  MsgKey      char(4);
end-Ds;

// Constants                                         3
Dcl-c ResumeNextMI    Const(10);
Dcl-c PercolCallStk  Const(20);
Dcl-c PercolNextHnd  Const(21);
Dcl-c PromoteCallStk Const(30);
Dcl-c PromoteNextHnd Const(31);

// Parameter data for CEENCOD procedure             4
Dcl-s MsgSeverity  int(5) Inz(4);
Dcl-s Case          int(5) Inz(1);
Dcl-s Severity      int(5) Inz(4);
Dcl-s Control       int(5) Inz(0);
Dcl-s FacilityID   char(3) Inz('USR');

// Start the program
dcl-proc IleErrhdl export;

// Input and output parameters                      5
Dcl-pi IleErrHdl;

```

```

CondToken  char(12);
ProcName   char(10);
RtnAct     int(10);
NewToken   char(12);
end-Pi;

```

Note the following points, which correspond to the numbers shown in the right side of Example 4-27 on page 127:

1. Parameters for the CEENCOD ILE API program are specified as a prototype in the /COPY member file ILEERRPR. The last parameter (the feedback data structure) is omitted.
2. The condition token data structure is shown.
3. Result codes are defined as constants.
4. Parameter data for the CEENCOD API program are initialized. MsgNumber and FacilityId variables are overwritten dynamically.
5. Parameters for the condition handler are given here and in the ILEERRPR copy member procedure.

The procedural part of the ILEERRHDL program tests for error messages and takes appropriate actions, as shown in Example 4-28.

Example 4-28 Program ILEERRHDL source

```

//=====
// Mainline program
//=====

CondTokenPtr = %addr(CondToken);

Select;

// Error RNX1218 Unable to allocate a record in file xxx... 1
When MsgType = 'RNX' and MsgNum = X'1218';
    ILECondHandler(MsgSeverity : MsgNum : Case :
        Severity : Control : FacilityID :
        MsgKey : NewToken : *Omit);
    RtnAct = PromoteCallStk;

// Error RNX1021 Attempt to write a duplicate record to file xxx... 2
When MsgType = 'RNX' and MsgNum = X'1021';
    ILECondHandler(MsgSeverity : MsgNum : Case :
        Severity : Control : FacilityID :
        MsgKey : NewToken : *Omit);
    RtnAct = PromoteNextHnd;

// Error MCH1211
// Attempt made to divide by zero for fixed point operation. 3
When MsgType = 'MCH' and MsgNum = X'1211';
    RtnAct = ResumeNextMI;
Other;                                         4
    RtnAct = PercolNextHnd;
EndSI;
Return;
end-Proc;

```

Note the following points, which correspond to the numbers shown in the right side of Example 4-28 on page 128:

1. If the incoming message is RNX1218 (a locked record), the condition handler decides to change it into a new message USR1218 and promote it to the next call stack entry. The new message is created by calling the CEENCOD ILE API program with parameters.
2. If the incoming message is RNX1021 (duplicate record), the condition handler changes the message into USR1021 and promotes it to the next call stack entry.
3. If the incoming message is MCH1211 (divide by zero), the condition handler tells the system to resume processing at the next instruction.
4. All the other messages are percolated to the next call stack entry handler.

ILE condition handler API prototypes

The file shown in Example 4-29 contains the ILE condition handler API prototype that is used in this example. It must exist before you create any of the modules that are used in this example.

Example 4-29 Prototype for the ILE condition handler

```

// ILEERRPR from ILESRC
// Input parameters
Dcl-pr IleErrHdl;
    CondToken  char(12);
    ProcName   char(10);
    RtnAct     int(10);
    NewToken   char(12);
end-Pr;
// Parameters for CEENCOD procedure (own message construction)
Dcl-pr ILECondHandler ExtProc('CEENCOD');
    MsgSeverity  int(5);
    MsgNumber    char(2);
    Case         int(5);
    Severity     int(5);
    Control      int(5);
    FacilityID  char(3);
    MsgKey       char(4);
    NewToken    char(12);
    Feedback     char(12) Options(*Omit);
end-Pr;

Dcl-pr RegCondHdlr ExtProc('CEEHDLR');
    CondHdlr@   pointer(*PROC);
    ProcName   char(10);
    Feedback   char(12) Options(*Omit);
end-Pr;

Dcl-pr UnRegCondHdlr ExtProc('CEEHDLU');
    CondHdlr@   pointer(*PROC);
    Feedback   char(12) Options(*Omit);
end-Pr;

```

Try it yourself: Create this module by running the following command:

```
CRTRPGMOD MODULE(RPGISCOOL/ILEERRHDL) SRCFILE(RPGISCOOL/ILESRC) SRCMBR(*MODULE)
```

This module is used in “Creating a program that uses the condition handler” on page 133.

A program that uses the condition handler

The program E01REG uses the condition handler ILEERRHDL. The program updates the STOCK database file with a check if the newly entered item exists in the ITEMS file. This action is done by calling the module E01ITEMS and checking the return code. The E01ITEMS module is bound by copy along with the E01REG and ILEERRHDL modules into the program E01REG. The module E01ITEMS contains an artificially generated error (divide by zero) to demonstrate how the condition handler (ILEERRHDL) handles messages in different modules.

Before the condition handler is used, it must be registered. This action is done by calling the CEEHDLR ILE API program with certain parameters.

When the condition handler is no longer needed, it can be unregistered by the CEEHDLU ILE API program. Only parts of the E01REG module are shown.

Note: The variable CondHdlr@ that is used in the E01REG module has broken the style guide rule found in 2.1.3.3, “Avoid using special characters (for example, @, #, \$) when naming items” on page 22. Avoid using the “@” symbol to indicate the variable is a pointer. A good method is to use the “Hungarian Notation,” where the first character of the variable name indicates the data type of the variable, such as pCustNbr or cActstsCde.

The data definition part of the E01REG module is shown in Example 4-30.

Example 4-30 Module E01REG source

```

//*****
// Member E01REG from ILESRC
//
// E01REG - Errors, condition handler registration
//
//*****


ctl-opt main(e01reg);

//=====
// File descriptions
//=====
dcl-F STOCK disk(*ext) keyed usage(*update : *output : *delete);

dcl-F STOCKW WorkStn;

// Program prototypes
/COPY ILESRC,ILEERRPR
/COPY ILESRC,e01itemspr

dcl-pr e01reg extpgm('E01REG');
end-Pr;

```

```

//=====
//  Data definitions
//=====

//  Procedure pointer to ILEERRHDL program
Dcl-s CONDHDLR@ pointer(*proc) Inz(%Paddr('ILEERRHDL')); 1

//  PSDS data structure (this procedure name)
Dcl-ds pgmDS psds NOOPT;
  ProcName *PROC; 2
end-Ds;

//  Key list for STOCK file
dcl-ds Key likerec(STOCKR : *key);

```

Note the following points, which correspond to the numbers shown in the right side of Example 4-30 on page 130:

1. The E01REG module defines a procedure pointer CondHdlr@ to the condition handler program (pointer to its main procedure). The name “ILEERRHDL” is an import symbol that is resolved into an address by binding the ILEERRHDL module (or a service program in which the module could be placed).
2. The current procedure name is taken from the PSDS data structure.
3. The CondHdlr@ pointer is specified as a first parameter and ProcName as the second parameter for the condition handler registration. The third parameter, feedback information, is omitted. The prototype parameters can be found in the /COPY member in “ILE condition handler API prototypes” on page 129.

To unregister the condition handler, only the CondHdlr@ pointer is needed. The prototype parameters can be found in the /COPY member in “ILE condition handler API prototypes” on page 129.

The relevant parts of the procedural part of the E01REG module are shown in Example 4-31.

Example 4-31 Program E01REG source

```

//=====
//  Mainline program
//=====

dcl-proc e01reg;
  dcl-pi e01reg;
end-Pi;

// local variables
dcl-s RC ind;

// Register an ILE condition handler program 1
RegCondHdlr(CondHdlr@ : ProcName : *Omit);

// Process stock data
...
  // Check item if present in the ITEMS file.
  // If present, RC is *OFF, if not present, RC is *ON.
  // A divide by zero error is artificially produced in E01ITEMS.
  rc = findItem(itemNbr); 2
...

```

```

// Read stock record and lock it for update.
// RNX1218 error message is sent to the program message queue
// if the record is locked by another job.
key STOCKNO = stockno;
key ITEMNBR = itemnbr;
Chain %kds(Key) STOCK;                                3
...
// Unregister the ILE condition handler program
UnregCondHdlr(CondHdlr@ : *0mit);                      4

```

Note the following points, which correspond to the numbers shown in the right side of Example 4-31 on page 131:

1. The E01REG module registers the condition handler.
2. After some processing, a check for the item number is made by calling the findItems procedure in the E01ITEMS module. In the module, a divide by zero error occurs (see the source for E01ITEMS in Example 4-32 on page 132). The condition handler practically ignores this message.
3. After some more processing, a record is read from the STOCK file for updating. If another job has locked the same record, the RNX1218 error message is generated by the system and sent to the program message queue. The condition handler handles it as required. It replaces the message by the USR1218 message, which has a different text.
4. The condition handler is unregistered.

For completeness Example 4-32 shows the E01ITEMS module.

Example 4-32 Module E01ITEMS source

```

ctl-opt nomain;
//*****
//
// Member E01ITEMS from ILESRC
// E01ITEMS - Check if an item is in ITEMS file
//
//*****
// prototypes
/COPY ILESRC,e0itemspr

// Start of find item procedure
dcl-proc findItem export;
  Dcl-pi findItem ind;
    itemNumber like(ITEMNBR);
  end-Pi;

//=====
// Data definitions
//=====
Dcl-s ReturnValue      ind;
Dcl-s Divisor         packed(9 : 2);
Dcl-s unitPrice       like(unitpr);
Dcl-s localItemNumber like(ITEMNBR);

// Look up the item number
// If not found - Set RC *ON else set RC *OFF

```

```

ReturnValue = *off;           // Assume found
localItemNumber = itemNumber;

exec sql
  select unitpr into :unitPrice
  from items
  where ITEMNBR = :itemNumber;

if sqlcode <> 0;
  ReturnValue = *on;
else;
  // Generate an artificial error (divide by zero)
  Divisor = 0;
  unitPrice = unitPrice / Divisor;
endif;

Return ReturnValue;

end-Proc findItem;

```

Try it yourself: You can create these two modules by using the two following commands. The two physical files and the display file that is used in this example must be created before creating the modules.

```
CRTRPGMOD MODULE(RPGISCOOL/E01REG) SRCFILE(RPGISCOOL/ILESRC) SRCMBR(*MODULE)
CRTRPGMOD MODULE(RPGISCOOL/E01ITEMS) SRCFILE(RPGISCOOL/ILESRC) SRCMBR(*MODULE)
```

These modules are used in “Creating a program that uses the condition handler” on page 133.

Creating a program that uses the condition handler

You can put these modules together in a number of ways. The major question is where to place the condition handler. The condition handler may be one that was written specifically for a particular application area, and in this case is not likely to be useful in other programs. It most likely is bound by copy into the program with the application modules.

However, in this example, the condition handler is fairly generic so that this same handler can be useful for many different programs. Therefore, it makes more sense to place the condition handler in a service program. In a real application environment, you most likely place it in a service program together with other modules, perhaps even other ILE condition handling modules. However, for purposes of simplicity in this example, the condition handler module is placed in a service program by itself. This service program is then bound by reference to the application program. The application program contains the application function modules bound together by copy.

The commands to create this scenario are shown here:

```
CRTSRVPGM SRVPGM(RPGISCOOL/SILEERRHDL) MODULE(RPGISCOOL/ILEERRHDL) EXPORT(*ALL)
ACTGRP(*CALLER)
```

```
CRTPGM PGM(RPGISCOOL/E01REG) MODULE(RPGISCOOL/E01REG RPGISCOOL/E01ITEMS)
BNDSRVPGM(RPGISCOOL/SILEERRHDL) ACTGRP(QILE)
```

Running a program that uses the condition handler

Call the program E01REG from the command line in a session. For this example, a record is locked while being displayed on the screen. Then, call the same program from another session and try to display the same record. After some time (60 seconds is the default), the second call ends with a message on the command line, as shown in Example 4-33.

Example 4-33 An error message after running a program with a condition handle

```
Application error. USR1218 unmonitored by E01REG at statement *N,
instruction X'0000'.
```

If you display the job log, the screen shown in Example 4-34 opens.

Example 4-34 Job log messages resulting from a program call with errors

```
> call e01reg
      Attempt made to divide by zero for fixed point operation. 1
      Record 4 in use by job or transaction 093172/DIEPHUIS/QPADEV0008.
? C
      Record 4 in use by job or transaction 093172/DIEPHUIS/QPADEV0008. 2
> s C
      Another program holds the same record. Wait please, or call the system
      administrator to find out who is it. 3
Application error. USR1218 unmonitored by E01REG at statement *N, 4
instruction X'0000'.
```

Note the following points which correspond to the numbers shown on the right side of Example 4-34 on page 134:

1. The divide by zero was generated by the FINDITEM procedure in module E01ITEMS in the E01REG program, as shown in Figure 4-12.

Message ID : MCH1211	Severity : 40
Date sent : 02/05/16	Time sent :
09:29:32	
Message type : Escape	
From : DIEPHUIS	CCSID : 65535
From program : E01REG	
From library : RADREDBOOK	
From module : E01ITEMS	
From procedure : FINDITEM	
From statement : 134	
To program : E01REG	
To library : RADREDBOOK	
To module : E01ITEMS	
To procedure : FINDITEM	
To statement : 134	

Figure 4-12 Details of the divide by zero message

2. Message ‘Record 4 in use...’ is sent by the system to module E01REG, as shown in Figure 4-13.

Message ID	:	CPF5027	Severity	:	30
Date sent	:	02/05/16	Time sent	:	
09:30:35					
Message type	:	Notify			
From	:	DIEPHUIS	CCSID	:	65535
From program	:	QDBSIGEX			
From library	:	QSYS			
Instruction	:	0272			
To program	:	E01REG			
To library	:	RADREDBOOK			
To module	:	E01REG			
To procedure	:	E01REG			
To statement	:	177			

Figure 4-13 Details of the record in use message

3. The message ‘Another job holds the same record...’ is the message USR1218 that replaced the message RNX1218 (which does not appear in the job log). Details are shown in Figure 4-14.

Message ID	:	USR1218	Severity	:	00
Date sent	:	02/05/16	Time sent	:	
09:30:35					
Message type	:	Escape			
From	:	DIEPHUIS			
From program	:	E01REG			
From library	:	RADREDBOOK			
From module	:	E01REG			
From procedure	:	E01REG			
From statement	:	177			
To program	:	E01REG			
To library	:	RADREDBOOK			
To module	:	E01REG			
To procedure	:	_QRNP_PEP_E01REG			
To statement	:	*N			

Figure 4-14 Details of the USR1218 message

4. The message ‘Application error...’ is the special *ESCAPE message sent by the system to the calling procedure QUOCMD above the control boundary of the QILE activation group. Figure 4-15 shows the message details.

```

Message ID . . . . . : CEE9901      Severity . . . . . : 30
Date sent . . . . . : 02/05/16      Time sent . . . . . :
09:30:35
Message type . . . . . : Escape
From . . . . . . . . . : DIEPHUIS    CCSID . . . . . . . . . : 65535

From program . . . . . . . . . : QLEAWI
From library . . . . . . . . . : QSYS
From module . . . . . . . . . : QLEDEH
From procedure . . . . . . . . . : Q LE 1eDefaultEh2
From statement . . . . . . . . . : 175

To program . . . . . . . . . : QCMD
To library . . . . . . . . . : QSYS
Instruction . . . . . . . . . : 01C8

```

Figure 4-15 Details of the CEE9901 message

Because no error was unhandled by the condition handler, no CPF9999 function check message appeared.

If you specified result code 10 (ResumeNextMI) for “other” messages in the condition handler, the job log is what is shown in Figure 4-16, for example:

RtnAct = [ResumeNextMI](#)

```

> call e01reg
Attempt made to divide by zero for fixed point operation.
Record 4 in use by job or transaction 093172/DIEPHUIS/QPADEV0008.
? C
Record 4 in use by job or transaction 093172/DIEPHUIS/QPADEV0008.
? C
Unable to allocate a record in file STOCK.
Duplicate record key in member STOCK.
Duplicate key not allowed for member STOCK.
? C
Duplicate key not allowed for member STOCK.
? C
You try to add a duplicate record. Review your program.
Function check. USR1021 unmonitored by E01REG at statement *N,
instruction X'0000'.

```

Figure 4-16 Job log with function check messages

Now, the function check escape messages are processed by the condition handler according to the rules in “Error handling in ILE” on page 123.

Try it yourself: To re-create a similar test case, you can use the following instructions. You need two sessions active on the same system. Be sure the physical files STOCK and ITEMS contain the sample data listed in the following section.

```
ADDLIB LIB(RPGISCOOL)
CALL PGM(RPGISCOOL/E01REG)
```

On the first screen, enter the following data:

```
Stock number: 00001
Item number: 00001
```

Press Enter. On another session, enter the same commands and the same data. After some time (60 seconds is default), the second call ends with a message on the command line, as shown in Example 4-33 on page 134.

Files used in the condition handling example

For completeness, here is the source of the files that are used in the conditional error handling examples. The STOCK and STOCKW files are used in programs E01REG.

Physical file description: STOCK

Example 4-35 shows the source code for the STOCK inventory file.

Example 4-35 Physical file description STOCK source

```
*****
* Member STOCK from ILESRC
* STOCK - Stock inventory file *
*****
A          UNIQUE
A          R STOCKR
* Data fields
A          STOCKNO      5      COLHDG('Stock' 'number')
A          ITEMNBR      5      COLHDG('Item' 'number')
A          QTYONHND    15 0   COLHDG('Quantity' 'onhand')
* Key fields
A          K STOCKNO
A          K ITEMNBR
```

Physical file content: STOCK

The following is a sample of the STOCK file, which is used with the running example:

Stock number	Item number	Quantity onhand
00001	00001	30

Display file STOCKW

Example 4-36 shows the STOCK inventory entry display file.

Example 4-36 Display file STOCKW source

```
*****
* Member STOCKW from ILESRC in RPGISCOOL *
* STOCKW - Stock inventory entry display file *
*****  

A DSPSIZ(24 80 *DS3)  

A CA03(03 'Exit')  

A CA12(12 'Cancel')  

* Format 1 - Prompt for the key  

A R STOCKW01  

A 3 4'Enter data.'  

A 5 4'Stock number:'  

A STOCKNO 5A B 5 22  

A 6 4'Item number:'  

A ITEMNBR 5A B 6 22  

A 7 22'Item does not exist'  

A 23 3'F3=Exit'  

A 23 16'F12=Cancel'  

* Format 2 - Display data from stock  

A R STOCKW02  

A CF23(23 'Delete')  

A 3 4'Stock inventory.'  

A 5 4'Stock number:'  

A STOCKNO 5A O 5 22  

A 6 4'Item number:'  

A ITEMNBR 5A O 6 22  

A 7 4'Quantity:'  

A QTYONHND 15Y 0B 7 22EDTCDE(Q)  

A 23 4'F3=Exit'  

A 23 15'F12=Cancel'  

A 23 30'F23=Delete'
```

Physical file ITEMS

The ITEMS file, shown in Example 4-37, is used in the E01ITEMS module.

Example 4-37 Database physical file ITEMS source

```
*****
* Member ITEMS from ILESRC
* ITEMS - Item master file
* A sample database physical file
*****  

A UNIQUE  

A R ITEMSPR  

* Item number  

A ITEMNBR 5 COLHDG('Item' 'number')  

* Unit price  

A UNITPR 9 2 COLHDG('Unit' 'price')  

* Item description  

A ITEMDESC 50 COLHDG('Item description')  

* Key field
```

A

K ITEMNBR

Try it yourself: Create the two physical files and the display file by running the following commands:

```
CRTPF FILE(RPGISCOOL/STOCK) SRCFILE(RPGISCOOL/IRESRC) SRCMBR(*FILE)
CRTPF FILE(RPGISCOOL/ITEMS) SRCFILE(RPGISCOOL/IRESRC) SRCMBR(*FILE)
CRTPRTF FILE(RPGISCOOL/REPORTS) SRCFILE(RPGISCOOL/IRESRC)
```

These files are used by the two programs that are described in “Creating a program that uses the condition handler” on page 133. The STOCK file can contain data that is specified in “Physical file content: STOCK” on page 137. An example of the contents of file ITEMS is listed in “File ITEMS” on page 116.

4.3 Additional CL commands and useful ILE APIs

Other means for working with modules, service programs, and programs are covered at a high level in 4.3.1, “Additional CL commands” and 4.3.2, “Some useful APIs to get information on ILE objects” on page 139.

4.3.1 Additional CL commands

Other commands and parameters may be useful if you are working on a larger project. The Update Program (**UPDPGM**) and Update Service Program (**UPDSRVPGM**) commands make it possible to create a version of a program or service program without having all the modules available. They replace selected modules in the program (or service program) with new versions of these modules without needing the other modules. Some consequences of this approach might be wanted. For more information about updating programs and service programs, see *IBM i Programming ILE Concepts*, SC41-5606.

The **OPTION(*UNRSLVREF)** parameter in the **CRTPGM** and **CRTSRVPGM** commands (and the **UPDPGM** and **UPDSRVPGM** commands) allows unresolved references (imports) in the program or service program. This parameter can be useful if you are testing a larger application. Some modules can refer to subprocedures that do not exist yet, but their interface is already known.

For more information about updating programs and service programs and unresolved references, see the *IBM i Programming ILE Concepts*, SC41-5606:

https://www.ibm.com/support/knowledgecenter/ssw_ibm_i_73/ilec/sc415606.pdf

4.3.2 Some useful APIs to get information on ILE objects

If you plan to write your own software to control changes in modules, service programs, and programs, you can use API programs that are available for ILE. You can find their descriptions in the API finder in the online IBM Knowledge Center for IBM i. Go to the “Program and CL Command” category. The following API programs might be especially useful:

- ▶ The List Module Information (QBNLMODI) API lists information about modules. The information is placed in a user space that is specified by you. This API is similar to the Display Module (**DSPMOD**) command. You can use the QBNLMODI API to do the following tasks:
 - List the defined symbols that can be exported to other modules.

- List the symbols that are defined externally to the module.
 - List procedure names and their type.
 - List objects that are referenced when the module is bound into an ILE program or service program.
 - List copyright information.
- 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 do the following tasks:
- List modules bound into a service program.
 - List service programs bound to a service program.
 - List data items that are 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 the procedure export information of a service program.
 - List data export information of a service program.
 - List signatures of a service program.
- 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 that is specified by you. You can use the QBNLPGMI API to do the following tasks:
- List modules that are bound into an ILE program.
 - List service programs that are bound to an ILE program.
 - List data items that are exported to the activation group.
 - List data item imports that are resolved by weak exports that were exported to the activation group.
 - List the copyrights of an ILE program.

You can, for example, list signatures of service programs that are bound in a program by using the QBNLSPGM API to get the “old” signatures. You can also list all “new” signatures of these service programs by using the QBNLPGMI API and compare the two lists to see if they match. If there is a mismatch, you can trigger a new binding of the program by using the **CRTPGM** command.

Be prepared to inspect lists of lists in some cases because the information that is retrieved by these APIs is organized hierarchically.

4.4 More information about ILE and shared open data paths

For more information about shared open data paths, see the IBM i Knowledge Center:

http://www-01.ibm.com/support/knowledgecenter/ssw_ibm_i/welcome

Once at the IBM Knowledge Center, click **Database** → **Programming**.

For more information, you can also consult these publications:

- ▶ There is essential information on ILE in *IBM i ILE Concepts*, SC41-5606, and *Rational Development Studio for i ILE RPG Programmer's Guide*, SC09-2507.
- ▶ *ILE Application Development Example*, SC41-5602 shows how to use ILE concepts in practice.
- ▶ ILE APIs, such as CEETREC for ending an activation group or CEEHDLR for registration a condition handler program and many more, are described in the IBM Knowledge Center API finder under the CEE category.
- ▶ You can use the IBM Knowledge Center API finder “Program and CL command” category to find APIs that are associated with ILE objects, especially for retrieving information about modules, service programs, and programs in user space.



Application programming interfaces

The application programming interfaces (APIs) on IBM i provide access to standard C library functions, standard Portable Operating System Interface (POSIX) functions, IBM i system interfaces, and even IBM i machine instructions (MI).

The following API topics are discussed in this chapter:

- ▶ 5.1, “Finding APIs” on page 144
- ▶ 5.2, “C functions” on page 145
- ▶ 5.3, “POSIX interfaces” on page 150
- ▶ 5.4, “IBM i system interfaces” on page 151
- ▶ 5.5, “Creating a reusable API” on page 155
- ▶ 5.6, “Things to remember” on page 162

5.1 Finding APIs

Because there are hundreds of APIs that are available on IBM i, the first problem is finding the ones that you want to use. The first place to start is the Application Programming Interfaces section of the IBM Knowledge Center for the IBM i release you are using.

The following link is for the Application Programming Interfaces section of the IBM i 7.2 Knowledge Center:

http://www.ibm.com/support/knowledgecenter/ssw_ibm_i_72/apiref/api.htm

Figure 5-1 shows the IBM Knowledge Center.

The screenshot shows the IBM Knowledge Center interface for IBM i 7.2. The left sidebar has a 'Table of Contents' section with a tree view. The 'Application programming interfaces' node is expanded, showing sub-sections like 'APIs overview', 'What's new for IBM i 7.2', 'PDF file for APIs', and 'API finder'. Below this, 'APIs by category' is expanded, listing various categories such as Backup and Recovery APIs, Client Management Support APIs, Cluster APIs, Communications APIs, Configuration APIs, Cryptographic Services APIs, Database and File APIs, Date and Time APIs, Debugger APIs, Dynamic Screen Manager APIs, Edit Function APIs, GDDM APIs, Hierarchical File System APIs, High-Level Language APIs, IBM HTTP Server for i APIs, IBM i Access for Windows APIs, IBM PASE for i APIs, ILE CEE APIs, International Components for Unicod, Java APIs, and Journal and Commit APIs. The main content area displays the 'Application programming interfaces' topic, which includes a brief introduction, a note about code examples, sections for 'APIs overview', 'What's new for IBM i 7.2', 'PDF file for APIs', 'API finder', 'APIs by category', 'Alphabetic list of APIs', 'API concepts', 'Using APIs', and 'Examples: APIs and exit programs'.

Figure 5-1 Application programming interfaces topic in the IBM i 7.2 Knowledge Center

There are various ways to find the specific API that you need. If you can make a reasonable guess as to how IBM classified the API, drilling down through the APIs by category can be useful. The API finder can be used if you know the API name or keywords that describe the API.

After you have found the API that you need, you must interpret the API description and either find or create the prototypes and possibly the data structures that are required to use the API. The C library and POSIX functions generally use C type prototypes and data structures, and the IBM i system interfaces are documented in a more language neutral way.

The QSYSINC library contains RPG IV source members that contain data structures and some prototypes that can be used in some cases. However, the data structures do not generally follow current RPG IV preferred practices, such as being qualified or using templates. In addition, prototypes for the actual procedure or program are not generally present. Fortunately, it is not difficult to create your own collection of source members after you understand how to map either the documentation or C include members to RPG IV.

5.2 C functions

The C programming language runtime environment includes many useful functions, including mathematical functions, and character and string handling. To use C functions from RPG IV, you must create prototypes for the C functions that you use. Some C functions also use specialized data structures for parameters, so you might need to create those too. Although RPG IV and C support the same data types, they are specified differently, as shown in Table 5-1.

Table 5-1 C to RPG IV data type mapping

C	RPG IV
int, long	int(10)
unsigned int, unsigned long	uns(10)
short	int(5)
unsigned short	uns(5)
long long	int(20)
unsigned long long	uns(20)
float	float(4)
double	float(8)
void *	pointer
char	char(1)
char[n]	char(n)

The QRPGLESRC file in the QSYSINC library contains member SYSTYPES, which contains a useful set of definitions that can be used to simplify the mapping of data types between RPG IV and C. For example, `i_int` is defined to be `int(10)`, so a field can be defined as `like(i_int)` instead of using Table 5-1 for conversion.

RPG IV and C also have different parameter passing conventions. The RPG IV default is to pass parameters by reference, and C passes parameters by value. To pass a parameter by value in RPG IV, the `value` keyword must be coded on the prototype. The RPG IV `options` keyword can also be used to assist in mapping parameter formats between RPG IV and C.

C function names often contain lower case characters, so the mixed case function name or the special value *dclcase must be provided by using the **extproc** keyword. For historical reasons, C by default automatically widens some parameter types when a procedure is called. The ***cwiden** option on the **extproc** keyword should be used to inform the RPG IV compiler that these parameter types should be widened.

Table 5-2 shows some example C to RPG IV parameter mappings.

Table 5-2 Example C to RPG IV parameter mappings

C	RPG IV	Notes
int parm1	parm1 int(10) value	None
int* parm2	parm2 int(10)	None
const char *parm3	parm3 pointer value options(*string:*trim)	Because the C parameter is defined as a constant, use option(*string:*trim) to trim a character field and convert it to a C string automatically.
char *parm4	parm4 char(n)	Because the C parameter is not defined as a constant, provide a buffer. The value of n depends on the size that is required by the C function.

5.2.1 Prototype mapping example

This section describes a prototype mapping example.

C math function

```
double sin ( double );
```

RPG IV prototype

```
dcl-pr sin float(8) extproc(*cwiden:*dclcase);
      x float(8) value;
end-pr;
```

Calling the function

```
dsply sin(0.524); // approximately pi/6 or 30 degrees
```

Result

```
DSPLY +5.003474302699141E-001
```

Notes:

- ▶ The C prototype can be found in QSYSINC/H(MATH). Although that prototype does not name the parameter, and it is possible to use *n instead of a name in the RPG IV prototype, naming the parameters generally makes prototypes more easily understood.
- ▶ Although the double type does not require widening, using the ***cwiden** option in prototypes for C functions reduces the risk of missing it when it is necessary.
- ▶ The ***dclcase** option tells the compiler that the actual name of the C function **sin** is in lower case instead of using the RPG IV default of converting function names to upper case. The actual function name of 'sin' in single quotes can be used as well.

5.2.2 A more complex prototype mapping example

This example uses the C regular expression functions to parse an email address from a character buffer.

RPG IV prototypes (CREGEXPR)

Figure 5-1 show the CREGEXPR RPG IV prototypes.

Example 5-1 CREGEXPR RPG IV prototypes

```
/IF DEFINED(CREGEXPR_Included)
  /EOF
/ENDIF
/DEFINE CREGEXPR_Included

// RPG IV types for common C types

/copy QSYSINC/QRPGLESRC,SYSTYPES

// regcomp() cflags

dcl-c REG_BASIC      0; // 0x0000 Basic RE rules (BRE)
dcl-c REG_EXTENDED   1; // 0x0001 Extended RE rules (ERE)
dcl-c REG_ICASE      2; // 0x0002 Ignore case in match
dcl-c REG_NEWLINE    4; // 0x0004 Convert <backslash><n> to <newline>
dcl-c REG_NOSUB      8; // 0x0008 regexec() not report subexpressions
dcl-c REG_ALT_NL     16; // 0x0010 POSIX: Use IFS newline

// regexec() eflags

dcl-c REG_NOTBOL    256; // 0x100 First character not start of line
dcl-c REG_NOTEOL    512; // 0x200 Last character not end of line

// Regular Expression error codes

dcl-c REG_NOMATCH   1; // RE pattern not found
dcl-c REG_BADPAT    2; // Invalid Regular Expression
dcl-c REG_ECOLLATE  3; // Invalid collating element
dcl-c REG_ECTYPE    4; // Invalid character class
dcl-c REG_EESCAPE   5; // Last character is \
dcl-c REG_ESUBREG   6; // Invalid number in \digit
dcl-c REG_EBRACK   7; // imbalance
dcl-c REG_EPAREN   8; // \( \) or () imbalance
dcl-c REG_EBRACE   9; // \{ \} or { } imbalance
dcl-c REG_BADBR   10; // Invalid \{ \} range exp
dcl-c REG_ERANGE   11; // Invalid range exp endpoint
dcl-c REG_ESPACE   12; // Out of memory
dcl-c REG_BADRPT   13; // ?*+ not preceded by valid RE
dcl-c REG_ECHAR    14; // invalid multibyte character
dcl-c REG_EBOL     15; // (shift 6 caret or not) anchor and not BOL
dcl-c REG_EEOL     16; // $ anchor and not EOL
dcl-c REG_ECOMP    17; // Unknown error in regcomp() call
dcl-c REG_EEXEC    18; // Unknown error in regexec() call
dcl-c REG_LAST     18; // Needs to always be the greatest
                      // regerror uses it to check for */
```

```

// valid number */
```

```

dcl-s C_LC_colval_t  like(i_int) template;
dcl-s C_mbstate_t    like(i_short);
dcl-s C_off_t         like(i_long);
```

```

dcl-ds C_regex_t      qualified template;
  re_nsub     like(size_t);
  re_comp     pointer;
  re_cflags   like(i_int);
  re_eroff    like(size_t);
  re_ucoll    like(C_LC_colval_t) dim(2);
  re_lsub     pointer;
  lsub_ar    like(size_t) dim(16);
  esub_ar    like(size_t) dim(16);
  *n         pointer;
  re_esub    pointer;
  re_specchar pointer;
  re_phdl    pointer;
  comp_spc   char(112);
  re_map     char(256);
  re_shift   like(C_mbstate_t);
  re_dbcs    like(i_short);
end-ds;
```

```

dcl-ds C_regmatch_t   qualified template;
  rm_so      like(C_off_t);
  rm_ss      like(C_mbstate_t);
  *n        like(i_short);
  rm_eo      like(C_off_t);
  rm_es      like(C_mbstate_t);
  *n        like(i_short);
end-ds;
```

```

// Constants to calculate maximum array size
```

```

dcl-c RPG_MAX_DATA_SIZE 16773104;
dcl-c RPG_MAX_REGMATCH  %div(RPG_MAX_DATA_SIZE:%size(C_regmatch_t));
```

```

dcl-pr regexec    like(i_int) extproc(*cwidens:*dclcase);
  preg      likeds(C_regex_t) const;
  string    pointer value options(*string);
  nmatch   like(size_t) value;
  pmatch   likeds(C_regmatch_t) dim(RPG_MAX_REGMATCH) options(*varsizes);
  eflags   like(i_int) value;
end-pr;
```

```

dcl-pr regcomp    like(i_int) extproc(*cwidens:*dclcase);
  preg      likeds(C_regex_t);
  string    pointer value options(*string);
  cflags   like(i_int) value;
end-pr;
```

```

dcl-pr regerror   like(size_t) extproc(*cwidens:*dclcase);
  errcode  like(i_int) value;
```

```

preg      likeds(C_regex_t) const;
errbuf    char(256) options(*varsize);
errbuf_size like(size_t) value;
end-pr;

dcl-pr regfree extproc(*cwidenc:*dclcase);
preg      likeds(C_regex_t);
end-pr;

```

Example program (XMPREGEX)

Figure 5-2 shows the XMPREGEX example program.

Example 5-2 XMPREGEX example program

```

ctl-opt main(main) dftactgrp(*no);

/copy cregexpr

// Main routine
dcl-prc main;
  dcl-pi main extpgm('XMPREGEX');
  end-pi;

  dcl-s email varchar(50); // Short enough for dspl operation

  email = getEmail('Rich Diedrich <rich@richdiedrich.com> ' +
                    'Rich Diedrich Consulting, LLC');
  dspl email;
end-prc;

// Find and return an email address from a string
dcl-prc getEmail;
  dcl-pi getEmail varchar(256);
    buffer varchar(8192) const options(*varsize);
  end-pi;

  // Simple pattern to capture an email address
  dcl-c pattern '([<@ ]+@[> ]+)';
  // Structure to hold compiled regular expression
  dcl-ds preg likeds(C_regex_t);
  // Array to hold offsets of matching string
  // Since we are only looking for a single address, we only need
  // one element
  dcl-ds pmatch likeds(C_regmatch_t) dim(1);
  dcl-s rc like(i_int);
  dcl-s matchval varchar(256) inz('');

  // Compile the pattern
  rc = regcomp(preg:pattern:REG_EXTENDED);
  if rc = 0;
    // Execute the regular expression against the input
    rc = regexec(preg:buffer:%elem(pmatch):pmatch:0);
    if rc = 0;
      // Set the result value as a substring of the input
      matchval = %subst(buffer:pmatch(1).rm_so + 1:

```

```

                pmatch(1).rm_eo - pmatch(1).rm_so);
        endif;
        // Free the memory used by the compiled pattern
        regfree(preg);
    endif;

    return matchval;
end-proc;

```

Result

Figure 5-2 shows the result of the XMPREGEX example program.

```
DSPLY rich@richdiedrich.com
```

Figure 5-2 Result of the XMPREGEX example program

Note: The original C prototypes are in QSYSINC/H(REGEX). The include member uses the system provided type mappings from QSYSINC/QRPGLESRC(SYSTYPES) for the mapping for the simple C types.

5.3 POSIX interfaces

The POSIX data structures and prototypes are generally presented in C syntax, therefore the same data type mappings are used. The POSIX standard defines interfaces for interacting with operating system functions, such as threads, file systems, and communication.

5.3.1 Accessing the IFS

The IBM i integrated file system (IFS) can be accessed through POSIX stream file interfaces. The IFS member of QRPGLESRC in the QSYSINC library contains a usable set of data definitions and prototypes, so you do not have to create those include members.

Example program (IFSXMP)

Figure 5-3 shows the IFSXMP example program.

Example 5-3 IFSXMP example program

```

ctl-opt main(main) dftactgrp(*no);

/include QSYSINC/QRPGLESRC,IFS

dcl-c UTF8_CCSID 1208;
dcl-c JOB_CCSID 0;

dcl-proc main;
  dcl-pi main extpgm('IFSXMP');
end-pi;

dcl-s filename varchar(100);
dcl-s line      varchar(100);
dcl-s fd        like(i_int); // File descriptor

```

```

dcl-s rc      like(i_int);

filename = '/richd/newfile.txt';
line = 'Test line' + STREAM_LINE_FEED; // Add line feed to text
rc = unlink(filename);           // Delete existing file
fd = open(filename:
            // Create an output file with a specified CCSID and
            // translate the characters when they are written
            O_CREAT + O_WRONLY + O_TEXTDATA + O_CCSID + O_TEXT_CREAT:
            // Permissions for output file
            S_IRWXU + S_IRGRP + S_IROTH:
            // Output file will be UTF-8, but we will be writing data
            // in the job CCSID
            UTF8_CCSID:JOB_CCSID);
// When writing from a varchar buffer, we need the address of
// the data
rc = write(fd:%addr(line:*data):%len(line));
rc = close(fd);
end-proc;

```

5.4 IBM i system interfaces

The documentation for IBM i system interfaces is done in a more language-neutral way, as shown in Table 5-3.

Table 5-3 System API to RPG IV data type mapping

System API	RPG IV
Char(n)	char(n)
Char(*)	char(n)1 or a data structure
Binary(4)	int(10)
Binary(4), Unsigned	uns(10)
Binary(2)	int(5)
Packed(m,n ^a)	packed(m:n)
PTR(SPP)	pointer

a. 'n' needs to be large enough to contain the expected value. If it refers to a data structure, the structure is described in the documentation.

Although the QSYSINC library contains data structure definitions for many of the system interfaces, they might not be usable when the structures contain fields of undefined length and they do not follow current RPG IV programming standards.

5.4.1 Standard techniques

Many of the system interfaces use standard structure definitions and programming techniques.

Error code parameter

One standard data structure is the error code parameter, ERRC0100, which is shown in Table 5-4.

Table 5-4 Error code parameter

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

Although the actual size of the exception data field is not defined, anything beyond the first 32,767 characters are not used. In practice, you can define the data structure to be much smaller than that.

Example 5-4 shows the error code parameter as an RPG IV template.

Example 5-4 Error code parameter as an RPG IV template

```
dcl-ds ERRC0100_t qualified template;
  Provided      int(10);
  Available    int(10);
  ExceptionID  char(7);
  *n           char(1);
  Data          char(48);
end-ds;
```

Note: The exception data (Data field) is defined to be 48 characters in this template. This is large enough in many cases.

List APIs

Because many system interfaces must return arbitrary length lists of information, a standard technique is to have the API return output data in a user space object that can be extended up to a little less than 16 MB. These APIs use a standard format for header data in the user space and specific data structures for list entries. The header data includes information about where the output data is in the user space, the number of list entries, and the size of each entry. You can use this information to move through the list by using data structures based on pointer values.

The header structures are documented in the API Concepts topic of the IBM i 7.2 Knowledge Center:

http://www.ibm.com/support/knowledgecenter/ssw_ibm_i_72/apiref/usfcommonstructure.htm

Example 5-5 shows a prototype template for generic header format 0100.

Example 5-5 Prototype template for generic header format 0100

```
dcl-ds GenHeader_0100_t qualified template;
  UserArea      char(64);
  GenHeaderSize int(10);
  ReleaseLevel  char(4);
  FormatName    char(8);
  APIUsed       char(10);
  DateTime      char(13);
  InfoStatus    char(1);
  SizeUsed      int(10);
  InputOffset   int(10);
  InputSize     int(10);
  HeaderOffset  int(10);
  HeaderSize    int(10);
  ListOffset    int(10);
  ListSize      int(10);
  ListCount     int(10);
  EntrySize     int(10);
  EntryCCSID   int(10);
  RegionId     char(2);
  LanguageId   char(3);
  Subsetted     char(1);
  *n            char(42);
end-ds;
```

User space APIs

To use a list API, you must create a user space, get a pointer to the data in the space, and possibly delete the space. The interfaces that are necessary to do this are documented in the User Space APIs topic in the IBM i 7.2 Knowledge Center:

http://www.ibm.com/support/knowledgecenter/ssw_ibm_i_72/apis/obj5.htm

Figure 5-6 shows a prototype for Create User Space (**QUSCRTUS**), Retrieve Pointer to User Space (**QUSPTRUS**), Delete User Space (**QUSDLTUS**).

Example 5-6 Prototype for Create User Space (QUSCRTUS)

```
//A prototype for Create User Space (QUSCRTUS)
dcl-pr quscr tus extpgm('QUSCRTUS');
  SpaceName      char(20) const;
  ExtendedAttribute char(10) const;
  InitialSize    int(10) const;
  initialValue   char(1) const;
  PublicAuthority char(10) const;
  TextDescription char(50) const;
  Replace        char(10) const options(*nopass);
  ErrorCode      char(32767) options(*varszie:*nopass);
  Domain         char(10) const options(*nopass);
  TransferSize   int(10) const options(*nopass);
  SpaceAlignment  char(1) const options(*nopass);
end-pr;

// A prototype for Retrieve Pointer to User Space (QUSPTRUS):
```

```

dcl-pr quspstrus extpgm('QUSPSTRU$');
  SpaceName      char(20) const;
  ReturnPointer   pointer;
  ErrorCode       char(32767) options(*varsize:*nopass);
end-pr;

// A prototype for Delete User Space (QUSDLTUS):

dcl-pr qusdltus extpgm('QUSDLTUS$');
  SpaceName      char(20) const;
  ErrorCode       char(32767) options(*varsize:*nopass);
end-pr;

```

5.4.2 List objects (QUSLOBJ) API

The list objects API uses the standard list technique to return information for a set of objects based on object name, library, and object type. The API can provide various levels of detail based on the specific format that is requested by the caller. This use of various formats that return different information is a common pattern in list APIs. For the QUSLOBJ API, the formats range from OBJL0100, which contains the name, library, and type to OBJL0700, which also includes usage, journaling, and ASP information.

For more information, see the List Objects (QUSLOBJ) API topic in the IBM i 7.2 Knowledge Center:

http://www.ibm.com/support/knowledgecenter/ssw_ibm_i_72/apis/quslobj.htm

This website contains detailed information about the API parameters, all the formats that are available, and descriptions of the data that is contained in each field of the data structures.

Example 5-7 shows the template for format OBJL0100.

Example 5-7 Prototype template for format OBJL0100

```

dcl-ds OBJL0100_t qualified template;
  ObjectName    char(10);
  LibraryName   char(10);
  ObjectType    char(10);
end-ds;

```

Example 5-8 shows the template for format OBJL0200.

Example 5-8 Prototype template for format OBJL0200

```

dcl-ds OBJL0200_t qualified template;
  ObjectName    char(10);
  LibraryName   char(10);
  ObjectType    char(10);
  InfoStatus    char(1);
  ExtendedAttr  char(10);
  TextDesc      char(50);
  UserAttr      char(10);
  *n            char(7);
end-ds;

```

Note: Fields marked as *Reserved* in the documentation do not need names, so you can use the special value of *n instead of coming up with names for multiple reserved fields.

Example 5-9 shows a prototype for the QUSLOBJ API.

Example 5-9 Prototype for the QUSLOBJ API

```
dcl-pr quslobj extpgm('QUSLOBJ');
  SpaceName  char(20) const;
  FormatName char(8) const;
  ObjectName char(20) const;
  ObjectType char(10) const;
  ErrorCode  char(32767) options(*varsize);
  Authority   char(238) const options(*varsize:*nopass);
  Selection    char(25) const options(*varsize:*nopass);
  ASPControl  char(24) const options(*varsize:*nopass);
end-pr;
```

Note: Parameters that are documented as Char(*) generally include the *varsize option, and optional parameters should include the *nopass option.

5.5 Creating a reusable API

Although the IBM i operating system provides hundreds of flexible APIs, you might want specific APIs for your application. These APIs can provide documented interfaces to your application that can be used by other parts of the application, by other applications within your business, or by other users of your application. These APIs can be made available as programs, procedures in service programs, or even as web services by using Integrated Web Services for IBM i.

The basic principles for creating a reusable API are that it should have the following characteristics:

- ▶ Encapsulated: The API should not expose access to internal variables except through documented interfaces.
- ▶ Stateless: The API should not keep an internal state. All of the information that is necessary to process a request should be in the parameters of the call itself.

For this example, use the system List Objects API to create a simple procedure that returns a list of objects with their types and descriptions that meet some user-specified selection criteria. The procedure is designed so that it can be easily used by the Integrated Web Services for IBM i tools.

When you create an API, you should create a source member that contains the data structures and prototypes that needed by a user of the interface.

Example 5-10 shows an example interface member.

Example 5-10 Example interface member

```
// Data structure for object information

dcl-ds ObjectInfo_t qualified template;
  Name      char(10);
  Library   char(10);
  Type      char(10);
  Attribute char(10);
  Description char(50);
end-ds;

// Procedure to return a simple list of objects
// Input parameters:
//  ObjectName - Either a name, a generic name, or *ALL
//  Library - Either a name, *CURLIB, *LIBL, *ALLUSR, *USRLIBL, or *ALL
//  ObjectType - Either an IBM i object type (like *FILE) or *ALL
// Output parameters:
//  ReturnCount - Number of entries in Objects array
//  Objects - Array of object information
// Return value:
//  0 or a value indicating the error

dcl-pr ListObjects int(10);
  ObjectName  char(10) const;
  Library    char(10) const;
  ObjectType char(10) const;
  ReturnCount int(10);
  Objects     likeds(ObjectInfo_t) dim(1000);
end-pr;
```

Note: The input fields are declared as const rather than value because you want this API to be usable by the Integrated Web Services for IBM i tools, which do not support value parameters that are not integers.

Example 5-11 shows the module source.

Example 5-11 Module source

```
// This module will be built as a service program, so no main procedure

// The interface information will be placed in the module, so the procedure
// can be easily be exposed as a web service through the IBM i IWS support

ctl-opt nomain pgminfo(*pcml:*module);

// Includes for API structures and prototypes
/include usecpr
/include qusspacepr
/include qusgenpr
/include quslobjpr

// Prototypes for the procedures to be available from this module
/include listobjpr

// Procedure will be exported so it is available outside of this module
dcl-proc ListObjects export;
  dcl-pi ListObjects int(10);
    ObjectName char(10) const;
    Library   char(10) const;
    ObjectType char(10) const;
    ReturnCount int(10);
    Objects    likeds(ObjectInfo_t) dim(1000);
  end-pi;

  dcl-ds SpaceName qualified;
    Name      char(10)  inz('LISTSPACE');
    Library   char(10)  inz('QTEMP');
  end-ds;

  dcl-ds ErrorCode likeds(ERRC0100_t);
  dcl-ds Header    likeds(GenHeader_0100_t) based(SpacePtr);
  dcl-ds Entry     likeds(OBJL0200_t) based(EntryPtr);
  dcl-s i         int(10);

// Initialize ErrorCode parameter
ErrorCode.Provided = %size(ErrorCode);
// Create user space in QTEMP for list
quscrtrus(SpaceName:'QUSLOBJ':1024:x'00':
           '*EXCLUDE':'Object List':'*YES':ErrorCode);
if ErrorCode.Available <> 0;
  return -1;
endif;
// Get a pointer to the user space
qusptrus(SpaceName:SpacePtr:ErrorCode);
if ErrorCode.Available <> 0;
  return -2;
endif;
// Fill the user space with the list of objects
quslobj(SpaceName:'OBJL0200':ObjectName + Library:ObjectType:
          ErrorCode);
```

```

if ErrorCode.Available <> 0;
    return -3;
endif;
// Get the number of objects found
ReturnCount = Header.ListCount;
// Limit the number returned to the array size
if ReturnCount > %elem(Objects);
    ReturnCount = %elem(Objects);
endif;
// Set the Entry data structure to the first element of the list
EntryPtr = SpacePtr + Header.ListOffset;
// Set the return array values
for i = 1 to ReturnCount;
    Objects(i).Name = Entry.ObjectName;
    Objects(i).Library = Entry.LibraryName;
    Objects(i).Type = Entry.ObjectType;
    Objects(i).Attribute = Entry.ExtendedAttr;
    Objects(i).Description = Entry.TextDesc;
    // Move the Entry data structure to the next element
    EntryPtr += Header.EntrySize;
endfor;
// Delete the user space
qusdltus(SpaceName:ErrorCode);
return 0;
end-proc;

```

After you write the code, create the module and the service program. Because you want your API to be exported from the service program, specify the procedure name in your export source member.

Example 5-12 shows the export source member.

Example 5-12 Export source member

```

STRPGMEXP PGMLVL(*CURRENT)
    EXPORT SYMBOL(LISTOBJECTS)
ENDPGMEXP

```

After you have your service program, add it to a binding directory so that it can be found and automatically bound into calling programs:

```

CRTBNDDIR BNDDIR(RADREDBOOK/MYAPI)
ADDBNDDIRE BNDDIR(RADREDBOOK/MYAPI) OBJ((LISTOBJ *SRVPGM))

```

Anyone that wants to use API must include the prototypes and use the binding directory, as shown in Example 5-13.

Example 5-13 Include the prototypes and the binding directory

```

// Use our API binding directory
ctl-opt main(main) dftactgrp(*no) bnddir('MYAPI');

// Include the prototypes
/include listobjpr

// Main procedure
dcl-proc main;

```

```
dcl-pi main extpgm('LISTOBJTST');
end-pi;

dcl-ds ObjectInfo likeds(0bjectInfo_t) dim(1000);
dcl-s count int(10);
dcl-s rc    int(10);
dcl-s i     int(10);

// Find all the service programs starting with L
rc = ListObjects('L*':'RADREDBOOK':'*SRVPGM':count:0bjectInfo);
if rc = 0;
  for i = 1 to count;
    dsply ObjectInfo(i).Name;
  endfor;
endif;
end-proc;
```

5.5.1 Making a web service

After you create a service program that contains your API and the PCML to describe the interface, you can use the Integrated Web Services for IBM i support to expose it as either a SOAP or REST web service.

The following website contains the documentation about how to use the wizards that are provided by Integrated Web Services for IBM i to deploy web services:

<http://www.ibm.com/systems/power/software/i/iws/>

For this example, see the information that is specific to your interface.

After navigating through a few pages to specify which service program you want to deploy as a SOAP services, you come to a page that shows your procedure parameters (Figure 5-3).

Select	Procedure name/Parameter name	Usage	Data type
<input checked="" type="checkbox"/>	▼ LISTOBJECTS		
	OBJECTNAME	input	char
	LIBRARY	input	char
	OBJECTTYPE	input	char
	RETURNCOUNT	input/output	int
	OBJECTS	input/output	struct

Buttons at the bottom: Select All, Deselect All, Expand All, Collapse All.

Figure 5-3 Procedure parameters

Because you defined the first three parameters as const, the wizard sets them as input parameters. The last two parameters default to input/output because there is no way to identify them as output only in your RPG IV prototype. Change them to output only and tell the wizard that the **RETURNCOUNT** parameter specifies the number of entries in the **OBJECTS** parameter, as shown in Figure 5-4.

Select	Procedure name/Parameter name	Usage	Data type	Count
<input checked="" type="checkbox"/>	▼ LISTOBJECTS			
	OBJECTNAME	input	char	
	LIBRARY	input	char	
	OBJECTTYPE	input	char	
	RETURNCOUNT	output	int	
	OBJECTS	output	struct	RETURNCOUNT

Buttons at the bottom: Select All, Deselect All, Expand All, Collapse All.

Figure 5-4 RETURNCOUNT specifying number of entries in the OBJECTS parameter

You navigate through a few more pages to finish deploying the service and test it.

Figure 5-5 shows the needed input parameters.

The screenshot shows a web-based configuration interface. At the top, there is a dropdown menu with the option 'listobjects'. Below it, another dropdown menu is open, showing 'arg0'. Under 'arg0', there are three input fields: 'LIBRARY' containing 'RADREDBOOK', 'OBJECTNAME' containing 'L*', and 'OBJECTTYPE' containing '*SRVPGM'. At the bottom of the form are two buttons: 'Go' and 'Reset'.

Figure 5-5 Input parameters

Click **Go**. The results are shown in Figure 5-6.

The screenshot shows a web-based response interface. At the top, there is a dropdown menu with the option 'listobjectsResponse'. Below it, another dropdown menu is open, showing 'return'. Under 'return', there is a section titled 'OBJECTS' containing several key-value pairs: 'ATTRIBUTE (string)': 'RPGLE', 'DESCRIPTION (string)': 'Example list objects API', 'LIBRARY (string)': 'RADREDBOOK', 'NAME (string)': 'LISTOBJ', 'TYPE (string)': '*SRVPGM', 'RETURNCOUNT (int)': '1', and 'returnValue (int)': '0'.

Figure 5-6 Results

You also can use the wizard to produce a REST web service. If you do that and enter the proper URL into our web browser:

`http://myserver/web/services/LISTOBJREST?library=RADREDBOOK&name=L*&type=*SRVPGM`

The response the comes back looks similar to Figure 5-7.

```
- <listobjectsResult>
  - <OBJECTS>
    <ATTRIBUTE>RPGLE</ATTRIBUTE>
    <DESCRIPTION>Example list objects API</DESCRIPTION>
    <LIBRARY>RADREDBOOK</LIBRARY>
    <NAME>LISTOBJ</NAME>
    <TYPE>*SRVPGM</TYPE>
  </OBJECTS>
  <RETURNCOUNT>1</RETURNCOUNT>
  <returnValue>0</returnValue>
</listobjectsResult>
```

Figure 5-7 REST web service response

5.6 Things to remember

Here are some things to remember regarding APIs:

- ▶ There are many APIs for a wide variety of functions that are available.
- ▶ Use the IBM Knowledge Center to explore the available APIs.
- ▶ The QSYSINC library contains prototypes for many of the APIs:
 - The C and POSIX function members can be used as a reference if there are not RPG IV members for the API that is needed.
 - The RPG IV members may not follow preferred practices.
- ▶ API prototypes should be placed in separate members so they can be reused.
- ▶ When creating new APIs, consider the following items:
 - They should be encapsulated and stateless.
 - Create prototype members.
 - Create export members.
 - Create binding directories.
 - If there is a possibility that the API will be exposed as a web service, include the PCML in the member.



Database access with RPG IV

Probably one of the most powerful aspects of RPG IV on IBM i is its ability to integrate with the database. This chapter reviews and provides examples of this tight integration between the language and database.

Over the years, a number of different ways to access databases on a wide variety of systems has evolved. The IBM i system is no different. This chapter uses RPG IV as the common denominator as you explore the database access methods.

The following database access topics are discussed in this chapter:

- ▶ 6.1, “Externalizing input and output” on page 164
- ▶ 6.2, “Embedded SQL” on page 178
- ▶ 6.3, “Stored procedures” on page 188
- ▶ 6.4, “DB2 call level interface” on page 200
- ▶ 6.5, “Trigger programs” on page 236
- ▶ 6.6, “Commitment control” on page 241
- ▶ 6.7, “A note about globalization” on page 246
- ▶ 6.8, “More information about database access with RPG IV” on page 246

Note: The idea of externalizing your input and output (I/O) to allow your RPG IV applications to adapt quickly to new database requirements has been around since the first edition of this book. This edition retains the section on externalizing from the first edition (6.1, “Externalizing input and output” on page 164 in this book). This edition updates the examples to be free form, but otherwise have only small changes in that section.

6.1 Externalizing input and output

Just what is meant by the term *externalizing input/output* (I/O)? It seems as though the files already are externalized? After all, they are defined with DDS rather than the old RPG II approach of defining files in input and output specifications.

This section explains externalizing I/O and shows you some of the benefits of taking this approach. Some of this information was described in the Chapter 2, “Programming RPG IV with style” on page 11

6.1.1 What is meant by externalizing

Externalizing means that all READ, CHAIN, and other database operations are in separate routines and service programs that require requests to these routines to perform the operation on the callers behalf.

Why externalize

Why would you want to externalize? The following example provides good reasons.

Suppose that during discussions with your users it becomes apparent to you that business needs have changed. After studying the new requirements for awhile, you realize that you must redesign the database to accommodate these changes.

Which of the following courses do you take:

- ▶ Modify your database and then locate all relevant I/O operations and modify them as required?
- ▶ Decide that doing the right thing is just too much work, and just "hack" the database one more time?

If you chose the second option, it might be because your knowledge of your applications tells you that the database I/O is spread liberally throughout the programs, which makes the impact of a database change difficult to estimate and equally difficult to implement. Do not feel embarrassed if you pick the second option. If you are entirely honest, this is the answer that 90% or more of all RPG users would give.

Many of the applications that you use today have evolved from an original System/36 base. Sometimes their history goes even further back in time to a System/3 or System/34. Even applications designed for the System/38 (the IBM i system's predecessor) were often forced to trade off design against performance. If your application is relatively new, it was probably created by duplicating portions of existing applications. Unfortunately, the benefits of this approach (mainly speed of development) tend to be offset by the disadvantages (perpetuation of problems). Of course it is easy to overlook the disadvantages when you are pressed for time to produce as quickly as possible, and you can always re-do it later.

The benefits of externalizing

Externalizing I/O operations provides one way of helping ensure that your applications can adapt quickly and (relatively) easily to changing business needs. Instead of coding READ, CHAIN, and so on, at each point in the program where database access is required, you invoke a routine to perform the I/O for you.

Although many people have successfully implemented such schemes in RPG/400, doing so requires that you make program calls to the I/O routines. By using the RPG IV subprocedures, you can now provide this interface in a much more natural language fashion. For example, a routine to read a record from the Customer Master file might be called ReadCustomer and can be designed to either read a record by key or simply return the next record in the sequence. It might look like the code that is shown in Example 6-1.

Example 6-1 Customer master file

```
// read by key  
custRecord = readCustomer(CustKey);  
// Read next record  
custRecord = readCustomer;
```

Alternatively, you might choose to use different names for the two functions, for example, ReadCustKey and ReadNextCust. If you need background information and details about the subprocedures, see Chapter 2, “Programming RPG IV with style” on page 11.

One step at a time

At first glance, it might seem that making these changes requires much work. The main point to remember is that you do not have to apply this technique to every database at the same time. Start with one that has been subject to change in the past and then work on others as time permits. One possible approach is to switch each database to the new method at the time you must make changes to it.

6.1.2 Putting theory into practice: An example of externalizing I/O

The following sections give you a brief and highly simplified example of using subprocedures to externalize your I/O. Hopefully by the time you finish these sections, you can see just how beneficial it can be. You might even have a few ideas about how you can apply the principles to your own applications.

The following sections lead you through this process:

- ▶ The first section (6.1.3, “Externalizing example: Overview” on page 166) explains the system in its initial state. For this example, the system consists of a single file and a single program. This example is intended to be only representative of the overall system.
- ▶ The next section (6.1.4, “Externalizing example: Separating database logic from display logic” on page 170) takes you through the steps that are required to move the file I/O operations into a separate subprocedure.
- ▶ The third section 6.1.5, “Externalizing example: Implementing changes” on page 174) shows you how the changes in the underlying database design can be accommodated with little or no impact to the rest of the system.
- ▶ The last section (6.1.6, “Externalizing example: Other possibilities” on page 177) describes other possible database changes and alternative methods for passing the record data between the procedures.

6.1.3 Externalizing example: Overview

The following sections describe the database, display file, and initial program for this example.

The display file CUSTDISP

All of the programs in this section use the same CUSTDISP display file shown in Figure 6-1.

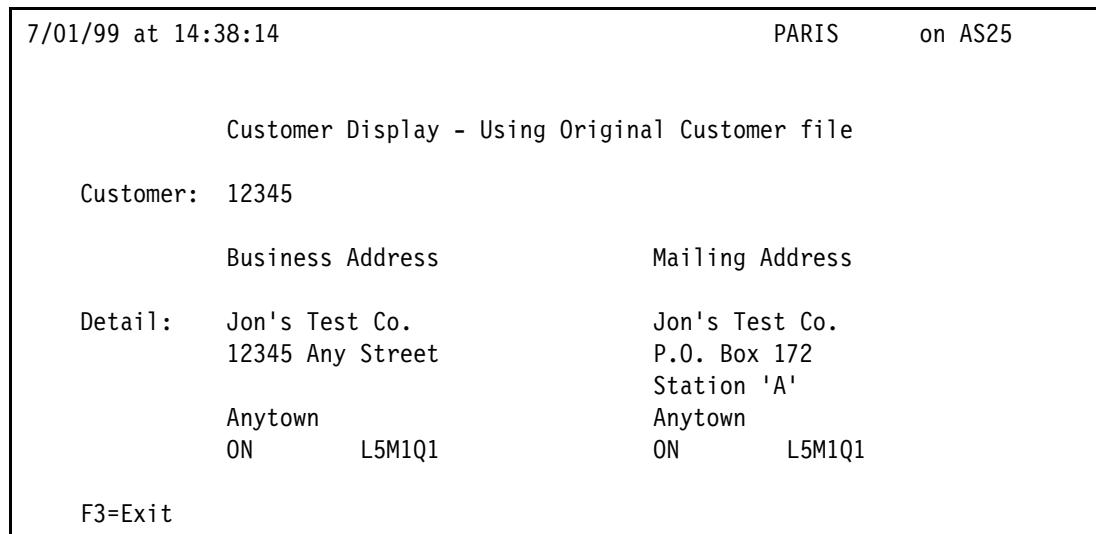


Figure 6-1 CUSTDISP display file

To use the program, enter a 5-character account number and the program attempts to retrieve the corresponding business address record. If the mailing address flag in the record indicates that there is a separate mailing address, that record is also retrieved and displayed. If either record is missing, an appropriate error message appears.

The text Original Customer file that appears on the sample display is supplied by the program. Each of the samples display a different value.

Example 6-2 shows the source code for the display file.

Example 6-2 Source code for the CUSTDISP display file

```

A* CUSTDISP from EXTIOSRC in RPGISCOOL
A                               DSPSIZ(24 80 *DS3)
A                               REF(PARIS/CUSTOMER CUSTDET)
A                               INDARA
A                               CF03(03)
A           R DISPREC
A                               1 2DATE
A                               EDTCDE(Y)
A                               1 11'at'
A                               1 14TIME
A                               1 59USER
A                               1 70'on'
A                               1 73SYSNAME
A                               4 16'Customer Display - Using'
A           VERSION      30A  0  4 41
A                               6 5'Customer:'
```

```

A      CUST#      R      B  6 16
A                      8 16'Business Address'
A                      8 48'Mailing Address'
A                      10 5'Detail:'
A      B_CUSNAME R    0 10 16REFFLD(CUSNAME)
A      B_ADDRESS1R   0 11 16REFFLD(ADDRESS1)
A      B_ADDRESS2R   0 12 16REFFLD(ADDRESS2)
A      B_CITY        R    0 13 16REFFLD(CITY)
A      B_PROVINCER   R    0 14 16REFFLD(PROVINCE)
A      B_POSTCODER   R    0 14 26REFFLD(POSTCODE)
A      M_CUSNAME R    0 10 48REFFLD(CUSNAME)
A      M_ADDRESS1R   0 11 48REFFLD(ADDRESS1)
A      M_ADDRESS2R   0 12 48REFFLD(ADDRESS2)
A      M_CITY        R    0 13 48REFFLD(CITY)
A      M_PROVINCER   R    0 14 48REFFLD(PROVINCE)
A      M_POSTCODER   R    0 14 58REFFLD(POSTCODE)
A                      16 5'F3=Exit'

```

The initial database design

The initial database design is a misnomer. It is a System/36 style flat file with multiple record types.

The file contains two distinct record types: Business address records and Mailing address records. The Business address records are indicated by a "B" in the field ADDRFLAG. The Mailing address records are indicated by an "M" in the same field.

There is a Business address record for each customer. Mailing address records are optional. If one exists, its presence is indicated by the appearance of an "M" in the MAILFLAG field of the corresponding Business address record.

Example 6-3 shows the DDS for the file.

Example 6-3 DDS source to define the DataBase

```

=====
* CUSTOMER from EXTOSRC in RPGISCOOL
* File Description: Customer Master File
* Key:          Customer No. & Mail Address Flag
*             CUST#     ADDRFLAG
=====
A      R CUSTDET           UNIQUE
A      CUST#      5A      TEXT('Customer #')
A      ADDRFLAG   1A      TEXT('M=Mail B=Business')
A      MAILFLAG   1A      TEXT('Separate Mail Addr')
A      CUSNAME    30A     TEXT('Customer Name')
A      ADDRESS1   30A     TEXT('Address 1')
A      ADDRESS2   30A     TEXT('Address 2')
A      CITY       20A     TEXT('City')
A      PROVINCE   3A      TEXT('Province')
A      POSTCODE   6A      TEXT('Postal Code')
A      K CUST#
A      K ADDRFLAG

```

Program SHOWCUST

The initial program in this example is SHOWCUST. When you enter a Customer number, it first retrieves the business address record. If the separate mailing address flag AddrFlag in the record is set to "M", it sets up the appropriate key value and attempts to retrieve the mailing address. It then displays the address data. If the Customer is not found, or if the mailing address is missing, an appropriate error message is displayed. For the sake of simplicity, error messages simply appears in the appropriate address fields.

Example 6-4 shows the program SHOWCUST source code.

Example 6-4 Program SHOWCUST source

```
// File SHOWCUST from EXTIOSRC in RPGISCOOL
//-----
// The information contained in this document has not been submitted
// to any formal IBM test and is distributed AS IS. 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. See Special Notices in rebook SG24-5402 for more.
//-----

dcl-f CustDisp workstn(*ext) IndDs(D_Indicators); 1

dcl-f Customer disk(*ext) usage(*input) keyed;

Dcl-ds D_Indicators;                                1
// Response Indicators
  Exit      ind pos(3);
end-ds;

// Constants used in the program
Dcl-c NotFound      'Error - Customer not found';
Dcl-c MailAddrErr   'Error - Mail address missing';
Dcl-c SameAsBus     'Use business address';
Dcl-c Business      'B';
Dcl-c Mailing        'M';
Dcl-c Separate       'M';

// External data structures used for database and display I/O 2
Dcl-ds CustAddr     ExtName(Customer);
Dcl-ds BusAddr       ExtName(Customer) Prefix(B_);
Dcl-ds MailAddr      ExtName(Customer) Prefix(M_);

dcl-ds custKey       likerec(custdet : *key);

Version = 'Original Customer file';

// Initial 'priming' read of display file
ExFmt DispRec;

DoU Exit;

// Set up key and search for Business address
// Note: Key list CustKey is defined above
AddrFlag = Business;
```

```

Chain %kds(CustKey) Customer;

// If the record is found set up display data and retrieve
// Mailing address if MailFlag indicates that one is available
If %Found(Customer);
  BusAddr = CustAddr;

  If MailFlag <> Separate;
    MailAddr = *Blanks;
    M_CusName = SameAsBus;
  Else;
    // Set up key and retrieve mailing address record
    AddrFlag = Mailing;
    Chain %kds(CustKey) Customer;

    If %Found(Customer);
      MailAddr = CustAddr;
    Else;
      // Error - no mailing record found - display error text
      MailAddr = *Blanks;
      M_CusName = MailAddrErr;
    EndIf;
  EndIf;

Else;
  // Requested Customer not found - display error text
  BusAddr = *Blanks;
  MailAddr = *Blanks;
  B_CusName = NotFound;
EndIf;
// Display results and accept next request
ExFmt DispRec;

EndDo;

*InLR = *On;
return;

```

Note the following points, which correspond to the numbers shown on the right side of Example 6-4 on page 168:

1. This example takes advantage of the Indicator Data Structure (INDDS) facility. By using INDDS, the programmer can assign names to response and conditioning indicators. The From/To notation method must be used to indicate which indicator is being defined. In this example, display file indicator 03 is associated with the name Exit.
The DDS keyword INDARA had to be specified on the display CUSTDISP to use this facility.
2. To simplify the moving of data from the file to the display screen, the Customer file record is specified as an externally described data structure CustAddr. Data structures are also defined for the Business address (BusAddr) and the Mailing address (MailAddr) by using the same external definition but using the Prefix keyword to generate different field names. A quick check of the display file shows you that these names are the ones that are used to display the data, which allows the file to be read and all relevant fields to be populated by simply moving the entire CustAddr DS to either BusAddr or MailAddr as appropriate.

Try it yourself: Complete the following steps:

1. Create the display file from the source:

```
CRTDSPF FILE(RPGISCOOL/CUSTDISP) SRCFILE(RPGISCOOL/EXTIOSRC)
SRCMBR(CUSTDISP)
```

2. Create the customer physical file:

```
CRTDSPF FILE(RPGISCOOL/CUSTOMER) SRCFILE(RPGISCOOL/EXTIOSRC)
SRCMBR(CUSTOMER)
```

Here is a partial screen capture with a sample of the data that is used to populate the file:

CUST#	ADDRFLAG	MAILFLAG	CUSNAME	ADDRESS1
12345	B	M	Jon's Test Co.	12345 Any Street
12345	M		Jon's Test Co.	P.O. Box 172
23456	B		MailSameAsBus Inc.	23456 The Street
34567	B	M	Mail Address Missing	34567 Lost Avenue

3. Create the program and the program call:

```
CRTPGM PGM(RPGISCOOL/SHOWCUST)
CALL RPGISCOOL/SHOWCUST
```

6.1.4 Externalizing example: Separating database logic from display logic

This section covers the steps that are involved with separating the database logic from the display logic, which sets you up for the subsequent database changes and provides you with database I/O routines that can be used, for example, from a Java client or even a Web Service.

The new structure

This section shows the creation of a new version of the main program SHOWCUST that is named SHOWCUST2. For the purposes of this example, the display file I/O is kept in this main program. It is so simple that there would be nothing left if you externalized that portion of the logic. However, in the real world, externalizing the display file I/O is useful because you can replace the 5250 screen with a modern web service or mobile interface.

The database I/O functions are moved to a new service program that is named GETCUST2. As you see later, after the separation is made, it becomes far easier to accommodate changes in the database design.

The GETCUST2 service program

A brief study of this code reveals that the logic itself is almost unchanged from the original code in SHOWCUST.

Example 6-5 shows the source code for the subprocedure GetCust.

Example 6-5 Source code for subprocedure GetCust

```
ctl-opt NoMain;
// GETCUST2 from EXTIOSRC in RPGISCOOL
//-----
// The information contained in this document has not been submitted
// to any formal IBM test and is distributed AS IS. 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. See Special Notices in redbook SG24-5402 for more.
//-----

dcl-f Customer disk(*ext) usage(*input) keyed;

/Copy RPGisCool/ExtIOSrc,GetCustPr          1

Dcl-c MailAddrErr  'Error - Mail address missing';
Dcl-c SameAsBus    'Use business address';
Dcl-c Business     'B';
Dcl-c Mailing      'M';
Dcl-c Separate     'M';

Dcl-c Found        *On;
Dcl-c NotFound     *Off;

// Data structure used when retrieving customer records
Dcl-ds CustAddr    ExtName('CUSTOMER');
end-Ds;

// Structures to pass business and mailing address data to caller
Dcl-ds BusAddr      ExtName('CUSTOMER') Prefix(B_) export;  4
end-Ds;
Dcl-ds MailAddr     ExtName('CUSTOMER') Prefix(M_) export;  4
end-Ds;

// key structure
dcl-ds custKey      likerec(custdet : *key);

dcl-proc GetCust   Export;
Dcl-pi GetCust ind ;
  CustNum Like(Cust#);
end-Pi;

// Set up key and search for Business address
AddrFlag = Business;
Chain %kds(CustKey) Customer;

// If the record is found set up data and retrieve
// Mailing address if MailFlag indicates that one is available
If %Found(Customer);
  BusAddr = CustAddr;

  If MailFlag <> Separate;
    MailAddr = *Blanks;
    M_CusName = SameAsBus;

  Else;
    AddrFlag = Mailing;
    Chain %kds(CustKey) Customer;

    If %Found(Customer);
      MailAddr = CustAddr;

```

```

Else;
  // Error - no mailing record found - set up error text
  MailAddr = *Blanks;
  M_CusName = MailAddrErr;

  EndIf;
EndIf;
// We have retrieved at least the Business address so return found
Return Found;          3

Else;
  // Requested Customer not found - set up error text and return
  BusAddr = *Blanks;
  MailAddr = *Blanks;
  B_CusName = NotFound;
  Return NotFound;          3
EndIf;
end-proc GetCust;

```

Note the following points, which correspond to the numbers shown on the right side of Example 6-5 on page 170:

1. You produce the prototype for the new subprocedure and added a /Copy statement to bring that source member into the program.
2. You add the procedure interface. If you read the previous chapter on ILE programming (Chapter 4, “An ILE guide for the RPG programmer” on page 770), this task should be familiar; if not, consider reading that chapter before continuing with this section because these features are not described here.
3. You added the Return op-code. In this example, it returns a named indicator (data type ind) to notify the caller of the success or failure of the read operation.
4. Because the purpose of the sub-procedure is to retrieve the Customer information on behalf of its caller, you find some way to accomplish this task. There are many different methods that you can use, but in this example, use the ILE Import/Export capability. The data that is read from the file is “exported” so that it can be “imported” by the main program, which allows the two procedures to share the data without needing for it to be passed as parameters. Some programmers do not like this approach, but in this limited context, it seems to be the method to use. This chapter later describes alternative methods that might be used.

In this example, the Import/Export data structures use the definitions of the underlying databases to simplify the example. In practice, you might choose to develop a separate composite definition for the express purpose of passing back the result set.

The main program SHOWCUST2

Program SHOWCUST2 is a modified version of the original SHOWCUST. Example 6-6 shows the source code for SHOWCUST2.

Example 6-6 Program SHOWCUST2 source

```

// File SHOWCUST2 from EXTIOSRC in RPGISCOOL
//-----
// The information contained in this document has not been submitted
// to any formal IBM test and is distributed AS IS. 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. See Special Notices in redbook SG24-5402 for more.
//-----
dcl-f CustDisp workstn(*ext) IndDs(D_Indicators);

/Copy RPGisCool/ExtIOSrc,GetCustPr          2

// Response Indicators
Dcl-ds D_Indicators;
  Exit    ind pos(3);
end-ds;

// Constants used in the program
Dcl-c True           *On;
Dcl-c NotFound       'Error - Customer not found';

Dcl-ds BusAddr ExtName('CUSTOMER') Prefix(B_) import;      3
end-Ds;

Dcl-ds MailAddr ExtName('CUSTOMER') Prefix(M_) Import;      3
end-Ds;

Version = 'External I/O - Version 1';

ExFmt DispRec;

DoU Exit;

// GetCust returns an indicator which is on if no record was found
If Not GetCust(Cust#);                                1
  B_CusName = NotFound;
EndIf;

ExFmt DispRec;

EndDo;

*InLR = True;
return;

```

Note the following points, which correspond to the numbers shown on the right side of Example 6-6 on page 172:

1. The first modification made was to remove the file I/O logic and replace it with an invocation of the GetCust subprocedure.
2. The next step was the addition of a /COPY statement to incorporate the prototype for GetCust that was developed earlier.
3. To allow the program to access the data retrieved by GetCust, the Import keyword was added to the definitions for the data structures BusAddr and MailAddr. Whenever GetCust places data into these structures, it is "visible" to the program.

Try it yourself: To re-create this example on your system, you must compile the subprocedure and the main program by running the following commands. You can create a service program from GETCUST2 and bind it to the SHOWCUST2 module, but bind them together for the sake of simplicity.

```
CRTRPGMOD MODULE(RPGISCOOL/GETCUST2)
CRTRPGMOD MODULE(RPGISCOOL/SHOWCUST2)
CRTPGM PGM(RPGISCOOL/SHOWCUST2) MODULE(RPGISCOOL/SHOWCUST2 RPGISCOOL/GETCUST2)
CALL RPGISCOOL/SHOWCUST2
```

6.1.5 Externalizing example: Implementing changes

Imagine that you decided that you must perform additional normalization on the database. You intend to separate the two types of records (Business and Mailing) into their own individual databases (CUSTOMERB and CUSTOMERM). The following sections describe the process of implementing these changes.

Database changes

For the sake of simplicity, the two new databases retain the original field names. The RPG IV PREFIX keyword allows you to rename the fields at the program level. Each of the new files is keyed only on the Customer number (CUST#).

As shown in Example 6-7, the source code for the CUSTOMERB file is almost identical to the original CUSTOMER file (see “The initial database design” on page 167) with the exception that the record format was renamed to CUSTDETB.

Example 6-7 DDS source for Database CUSTOMERB

```
*=====
* CUSTOMERB from EXTIOSRC in RPGISCOOL
* File Name:          CUSTOMERB
* File Description:   Customer Business Addresses
* Key:                Customer Number (CUST#)
*=====

A          UNIQUE
A          R CUSTDETB
A          CUST#      5A      TEXT('Customer #')
A          MAILFLAG   1A      TEXT('M = Mailing Address')
A          CUSNAME    30A     TEXT('Customer Name')
A          ADDRESS1   30A     TEXT('Address 1')
A          ADDRESS2   30A     TEXT('Address 2')
A          CITY        20A     TEXT('City')
A          PROVINCE   3A      TEXT('Province')
A          POSTCODE   6A      TEXT('Postal Code')
A          K CUST#
```

The Mailing address file is similar. However, the ‘Separate mailing address’ flag was removed and its record format changed to CUSTDETM. Example 6-8 shows the source code.

Example 6-8 DDS source for DataBase file CUSTDETM

```
*=====
* CUSTOMERM from EXTIOSRC in RPGISCOOL
*   File Name:          CUSTOMERM
*   File Description:   Customer Mailing Addresses
*   Key:                Customer Number (CUST#)
*=====
A          UNIQUE
A          R CUSTDETM
A          CUST#      5A      TEXT('Customer #')
A          CUSNAME    30A     TEXT('Customer Name')
A          ADDRESS1   30A     TEXT('Address 1')
A          ADDRESS2   30A     TEXT('Address 2')
A          CITY        20A     TEXT('City')
A          PROVINCE   3A      TEXT('Province')
A          POSTCODE   6A      TEXT('Postal Code')
A          K CUST#
```

Changes to GetCust

GETCUST3 contains a modified version of the original subprocedure GetCust. It now retrieves the customer data by accessing the two separate databases.

Example 6-9 shows the modified source.

Example 6-9 Program GETCUST3 source

```
ctl-opt NoMain;
// source member GETCUST3 from EXTIOSRC in RPGISCOOL
//-----
// The information contained in this document has not been submitted
// to any formal IBM test and is distributed AS IS. 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. See Special Notices in redbook SG24-5402 for more.
//-----

dcl-f CustomerB disk(*ext) usage(*input) keyed prefix(B_); 1
dcl-f CustomerM disk(*ext) usage(*input) keyed prefix(M_);

/Copy RPGisCool/ExtIOSrc,GetCustPr

Dcl-c MailAddrErr 'Error - Mail address missing';
Dcl-c SameAsBus  'Use business address';
Dcl-c Business    'B';
Dcl-c Mailing    'M';
Dcl-c Separate   'M';
Dcl-c Found      *On;
Dcl-c NotFound   *Off;

// Structures to pass business and mailing address data to caller
Dcl-ds BusAddr    ExtName('CUSTOMER') Prefix(B_) export; 3
```

```

end-Ds
Dcl-ds MailAddr      ExtName('CUSTOMER') Prefix(M_) export;
end-Ds;

dcl-proc GetCust   Export;
Dcl-pi GetCust ind ;
  CustNum  Like(B_Cust#);          2
end-Pi;

Chain CustNum CustomerB;           4

// If the record is found set up data and retrieve
// Mailing address if MailFlag indicates that one is available
If %Found(CustomerB);

  If B_MailFlag <> Separate;
    MailAddr = *Blanks;
    M_CusName = SameAsBus;

  Else;
    Chain CustNum CustomerM;           5

    If %Found(CustomerM);
    Else;
      // Error - no mailing record found - set up error text
      MailAddr = *Blanks;
      M_CusName = MailAddrErr;

    EndIf;
  EndIf;
  // We have retrieved at least the Business address so return found
  Return Found;

Else;
  // Requested Customer not found - set up error text and return
  BusAddr = *Blanks;
  MailAddr = *Blanks;
  B_CusName = NotFound;
  Return NotFound;
EndIf;

end-proc GetCust;

```

Note the following points, which correspond to the numbers shown on the right side of Example 6-9 on page 175:

1. The first change was to modify the F specs to remove the old customer file and introduce the new Business and Mailing address files. The same Prefix entry is used later for the data structures BusAddr and MailAddr so that the data from each file is read directly into its associated structure 3.
2. The definition of CustNum in the procedure interface parameter list was modified to reference the field B_Cust# because the field Cust# that it referenced in GETCUST2 does not exist in this version.

Note: You might be tempted to go back and use B_Cust# as the reference field in GETCUST2 to avoid this change. That would have been cheating. Looking back to solving your problem is always easier. You will undoubtedly encounter similar "why didn't I think of that" situations in your own efforts.

3. The names of the new files are used to supply the external descriptions and the Prefix is used to match the field names to the files.
4. The CHAIN operation was modified to use CustNum as the key because there is no longer a need for a keylist. The key list and the EVAL that set up the ADDRFLAG field were removed. The customer records are now being read directly into their data structures, so there is no need to move the data after the read.
5. Similar changes were made to the code handling the read of the mailing address.

Changes to SHOWCUST

The time has come to test the claim that having externalized the I/O minimizes the impact of the changes to the calling programs even though you have only one calling program.

The only change that is required to produce SHOWCUST3 from the previous version was that the external definitions for BusAddr and MailAddr were changed to use the new database formats. That is the total extent of the changes as shown in Example 6-10.

Example 6-10 Changes to SHOWCUST3

```
Dcl-ds BusAddr ExtName('CUSTOMERB') Prefix(B_) import;
end-Ds;

Dcl-ds MailAddr ExtName('CUSTOMERB') Prefix(M_) import;
end-Ds;
```

If you used a different way of passing the information to the program, such as passing the business address and mailing address structures as parameters on the procedure originally, you could not make any changes to the calling program.

Try it yourself: To re-create this example on your system, you must compile the subprocedure and the main program by running the following commands:

```
CRTRPGMOD MODULE(RPGISCOOL/GETCUST3)
CRTRPGMOD MODULE(RPGISCOOL/SHOWCUST3)
CRTPGM PGM(RPGISCOOL/SHOWCUST3) MODULE(RPGISCOOL/SHOWCUST3 RPGISCOOL/GETCUST3)
CALL RPGISCOOL/SHOWCUST3
```

6.1.6 Externalizing example: Other possibilities

Suppose that you decide that the database will be further normalized by replacing the fields CITY and PROVINCE with a City code, which would reference a separate City database. What must you do to achieve this task?

This book does not provide a solution to this particular problem, so consider it as an exercise for you. However, when you create the new Business address database, use a new name. Retain the existing definition so that it can still be used to describe the Import/Export data structure BusAddr. Handle the new Mailing address database the same way.

6.1.7 Summary

Hopefully, by now you are convinced that externalizing your I/O can reduce the effort that is required to implement design changes in your database. It can also offer other, perhaps unexpected, benefits.

For example, with the I/O operations spread throughout the code, you probably never need to add additional logic to your programs to determine whether the record being requested already was read and locked by the previous request. If all I/O for the file is in one place, such a change is trivial, so this has the potential to offer significant performance benefits, particularly in a batch environment.

6.2 Embedded SQL

Instead of using traditional IBM i native database file operations, such as READ, CHAIN, UPDATE, and DELETE, you can embed SQL statements directly in the RPG IV program and use them to process records in the IBM i database files. SQL is the industry standard for database access and control, and is used by customers on many different platforms. The reasons for using embedded SQL in RPG IV programs on the IBM i system could be any of the following ones:

- ▶ Developers with SQL knowledge can write RPG programs without learning native file operations.
- ▶ SQL is a more natural language and such code is easier to read and maintain.
- ▶ SQL can simplify the program logic when multiple records are included in an operation, such as UPDATE or DELETE. See the later section on using a cursor.
- ▶ SQL operations are performed by a query optimizer, which is enhanced with each new release, and automatically takes advantages of new database technologies.
- ▶ Today's modern developers understand SQL and can easily learn programming on IBM i when combined with modern RPG (full free-form).

Source code that contains embedded SQL statements must first be processed by an SQL preprocessor. Its job is to replace SQL statements with calls to corresponding SQL function programs. This preprocessor is a part of the IBM licensed product DB2 Query Manager and SQL Development Kit for IBM i (5770ST1), which must be available during application development. The runtime support is included in the operating system.

The request for additional chargeable software could be a reason for not using embedded SQL. In that case, you can try to use Call Level Interface APIs, described in 6.4.2, “Writing a DB2 CLI application” on page 202. These system APIs allow the use of SQL statements in RPG IV program, without needing an SQL preprocessor.

6.2.1 Rules for embedding SQL statements

Use the following rules when writing an RPG IV program with embedded SQL statements:

- ▶ Enter your SQL statements as calculation specifications.
- ▶ Start your SQL statements with the delimiter EXEC SQL.
- ▶ You can start entering your SQL statements on the same line as the starting delimiter or on the new line.
- ▶ Use a comma to end the SQL statement.

Example 6-11 shows an example of an embedded SQL **UPDATE** statement.

Example 6-11 An embedded SQL UPDATE statement

```
Exec Sq1
Update Parts
  Set PartDes = :DspDes,
      PartQty = :DspQty,
      PartPrc = :DspPrc,
      PartDat = :DspDat
  Where PartNum = :DspNum;
```

The source member containing the RPG IV program with embedded SQL statements must be of type SQRPGLE. Rational Developer for i gives you the option to call the **CRTSQLRPGI** CL command, which is required to call the SQL preprocessor.

6.2.2 SQL preprocessor

The SQL preprocessor creates an output source file member. By default, it creates a temporary source file called QSQLTEMP1 in the library QTEMP, which is automatically deleted by the system at the end of the job. You can specify the output source file as a permanent file name on the preprocessor command. A member with the same name as the program name is added to the output source file.

This member contains the following items:

- ▶ Calls to the SQL runtime support, which replaced embedded SQL statements.
- ▶ Parsed and syntax-checked SQL statements.

By default, the precompiler calls the host language compiler by using either the Create Bound RPG Program (**CRTBNDRPG**) or Create RPG Module (**CRTRPGMOD**) command, depending on the object value that is specified on the compile type parameter (**OBJTYPE**) of the create SQL ILE RPG object command (**CRTSQLRPGI**).

6.2.3 Error and exception handling

SQL does not communicate directly with the user, but returns error codes to the application program when an error or exception occurs. These error codes can be used in two ways:

- ▶ Checking return codes in an SQL Communication Area
- ▶ Defining global error handling by a **WHENEVER** statement

SQL communication area

The SQL preprocessor automatically includes the SQLCA (SQL Communication Area) in the data specifications of the RPG IV program before the first calculation specification. Therefore, it is not necessary to code **INCLUDE SQLCA** in the source program.

The SQLCA, included in ILE RPG program, contains the fields shown in Example 6-12.

Example 6-12 SQLXCA fields

D SQLCA	DS	
D SQLCAID		8A INZ(X'0000000000000000')
D SQLAID		8A OVERLAY(SQLCAID)
D SQLCABC		10I 0
D SQLABC		9B 0 OVERLAY(SQLCABC)

D SQLCODE	10I 0
D SQLCOD	9B 0 OVERLAY(SQLCODE)
D SQLERRML	5I 0
D SQLERL	4B 0 OVERLAY(SQLERRML)
D SQLERRMC	70A
D SQLERM	70A OVERLAY(SQLERRMC)
D SQLERRP	8A
D SQLERP	8A OVERLAY(SQLERRP)
D SQLERR	24A
D SQLER1	9B 0 OVERLAY(SQLERR:*NEXT)
D SQLER2	9B 0 OVERLAY(SQLERR:*NEXT)
D SQLER3	9B 0 OVERLAY(SQLERR:*NEXT)
D SQLER4	9B 0 OVERLAY(SQLERR:*NEXT)
D SQLER5	9B 0 OVERLAY(SQLERR:*NEXT)
D SQLER6	9B 0 OVERLAY(SQLERR:*NEXT)
D SQLERRD	10I 0 DIM(6) OVERLAY(SQLERR)
D SQLWRN	11A
D SQLWNO	1A OVERLAY(SQLWRN:*NEXT)
D SQLWN1	1A OVERLAY(SQLWRN:*NEXT)
D SQLWN2	1A OVERLAY(SQLWRN:*NEXT)
D SQLWN3	1A OVERLAY(SQLWRN:*NEXT)
D SQLWN4	1A OVERLAY(SQLWRN:*NEXT)
D SQLWN5	1A OVERLAY(SQLWRN:*NEXT)
D SQLWN6	1A OVERLAY(SQLWRN:*NEXT)
D SQLWN7	1A OVERLAY(SQLWRN:*NEXT)
D SQLWN8	1A OVERLAY(SQLWRN:*NEXT)
D SQLWN9	1A OVERLAY(SQLWRN:*NEXT)
D SQLWNA	1A OVERLAY(SQLWRN:*NEXT)
D SQLWARN	1A DIM(11) OVERLAY(SQLWRN)
D SQLSTATE	5A
D SQLSTT	5A OVERLAY(SQLSTATE)

The SQLCOD and SQLSTT values are set by the database manager after each SQL statement is run. A program should check either the SQLCOD or SQLSTT value to determine whether the last SQL statement was successful:

If the SQL encounters an error while processing the statement, the SQLCOD is a negative number, and the first two characters of the SQLSTT are not "00", "01", or "02".

If SQL encounters a warning but a valid condition while processing your statement, the SQLCOD is a positive number and the first two characters of the SQLSTT are "01".

If the SQL statement is processed without encountering an error or warning condition, the SQLCOD returned is 0 and SQLSTT is '00000'.

The commonly used condition "No record found" returns the value SQLCOD = +100 or SQLSTT = '02000'.

The Communication Area contains many fields with specific information relating to the executed SQL statement.

The WHENEVER statement

As an alternative to checking the SQLCOD or SQLSTT values, a programmer can use the SQL statement **WHENEVER**.

The **WHENEVER** statement causes SQL to check the SQLSTT and SQLCOD variable values and continue processing your program or branch to another area in your program if an error, exception, or warning exists as a result of running an SQL statement. An exception condition handling subroutine, written by a programmer, can then examine the SQLCOD or SQLSTT field to take an action specific to the error or exception situation.

The **WHENEVER** statement is shown in Example 6-13.

Example 6-13 WHENEVER code example

```
exec sql
  whenever condition action;
```

There are three conditions you can specify:

SQLWARNING	SQLCOD contains a positive value other than 100.
SQLERROR	SQLCOD contains a negative value (error condition).
NOT FOUND	SQLCOD = +100 or SQLSTT = '02000' (no record found).

You can also specify the action you want for a specific condition:

CONTINUE	Program continues to the next statement.
GO TO label	Program branches to a label (TAG) in the program.

6.2.4 Using a cursor

Opposite to native database operations, which are single record oriented and able to process only one record at the time, SQL statements are multiple-record oriented and can handle a group of records all at once. For example, with one **DELETE** statement, you can delete all item records for one order. Or, with one **UPDATE** statement, you can update all records in the file if the **WHERE** condition is not used. To achieve the same results with native file operations, you need to write program loops and test different conditions. Therefore, using SQL statements can often simplify the program logic.

According to this behavior, a **SELECT** statement puts all selected records in the result table. Usually, a program has to transfer all these records from an SQL result table to a subfile so the user can see them. To access a result table, SQL provides a technique called a *cursor*. It is used within an SQL program to maintain a position in the result table. SQL uses a cursor to work with the rows in the result table and to make them available to the program. A program can have several cursors, although each must have a unique name.

Here are statements that are related to using a cursor:

- ▶ The **DECLARE CURSOR** statement defines the name of the cursor and specifies the rows to be retrieved with the embedded **SELECT** statement.
- ▶ The **OPEN** statement opens the cursor for use within the program. The cursor must be opened before any rows can be retrieved.

- ▶ The **FETCH** statement retrieves rows from the cursor's result table or positions the cursor on another row.
- ▶ The **CLOSE** statement closes the cursor.

The code snippets shown in Example 6-14 illustrate the use of a cursor.

Example 6-14 Cursor code snippets

```
Exec Sql
  Declare C1 Cursor For
    Select * From Parts
      Order by PartNum
        For Fetch Only;

  Exec Sql
    Open C1;

  Exec Sql
    Fetch C1 Into :PartNo, :PartDs, :PartQy, :PartPr, :PartDt;

  Exec Sql
    Close C1;
```

SQL supports two types of cursors: serial and scrollable. The type of cursor determines the positioning methods that can be used with the cursor.

Serial cursor

A serial cursor is defined by default if the keyword **SCROLL** is not used. With a serial cursor, each row of the result table can be fetched only once per **OPEN** of the cursor. When the cursor is opened, it is positioned before the first row in the result table. With each **FETCH** statement, the cursor is moved to the next row in the result table, which becomes the current row. If host variables are specified (with the **INTO** clause on the **FETCH** statement), SQL moves the current row's contents into the program's host variables.

This sequence is repeated each time a **FETCH** statement is issued until the end-of-data (SQLCOD = 100) is reached. When you reach the end-of-data condition, close the cursor. You cannot access any rows in the result table after you reach the end-of-data condition. To use the cursor again, you must first close the cursor and then re-issue the **OPEN** statement.

Scrollable cursor

With a scrollable cursor, the rows of the result table can be fetched many times. The cursor is moved through the result table based on the position option that is specified on the **FETCH** statement. When the cursor is opened, it is positioned before the first row in the result table. With a **FETCH** statement, the cursor is positioned to the row in the result table that is specified by the position option. That row becomes the current row.

The following scroll options, relative to the current cursor location in the result table, are used to position the cursor when issuing a **FETCH** statement:

NEXT	Positions the cursor on the next row. The default if no position is specified.
PRIOR	Positions the cursor on the previous row.
FIRST	Positions the cursor on the first row.
LAST	Positions the cursor on the last row.

BEFORE	Positions the cursor before the first row.
AFTER	Positions the cursor after the last row.
CURRENT	Does not change the cursor position.
RELATIVE <i>n</i>	Positions the cursor for <i>n</i> rows relative to the current position.

6.2.5 An embedded SQL program example

To illustrate the coding of embedded SQL statements in RPG IV program, this section shows a simple example of a file maintenance program, where you can implement different SQL statements (**SELECT**, **INSERT**, **UPDATE**, and **DELETE**).

The program uses the display file DSPFIL1 as an interface with a terminal user and handles records in the database file PARTS. To help you understand the logic of a program, this example provides the DDS definitions for both the database and display file.

The database file PARTS (Example 6-15) contains five fields, with one of them used as a key.

Example 6-15 DDS for the PARTS database file

```

A*****
A* Physical file PARTS IN FILE DBSRC IN LIB RPGISCOOL
A*****
A          UNIQUE
A      R PARTR
A      PARTNUM      5S 0      COLHDG('Part Number')
A      PARTDES     25       COLHDG('Part Description')
A      PARTQTY      5P 0      COLHDG('Part Quantity')
A      PARTPRC     6P 2      COLHDG('Part Price')
A      PARTDAT      L       COLHDG('Shipment Date')
A          DATFMT(*ISO)
A      K PARTNUM

```

The display file DSPFIL1 (Example 6-16) contains two records and a subfile.

Example 6-16 Display file DSPFIL1 source

```

A*****
A* Display file DSPFIL1 IN FILE DBSRC IN LIB RPGISCOOL
A*****
A          INDARA
A          CA03(03 'Exit')
A          CA12(12 'Cancel')
A      R DSPREC1
A          CA04(04 'List All')
A          CF05(05 'Insert')
A          2 2'Enter part number:'
A      PARTNO      5Y 0I      +1
A      55           4 2'Invalid part number'
A          24 2'F3 = Exit F4 = List all'
A          +2'F5 = Insert F6 = Update'
A          +2'F7 = Delete F12 = Cancel'
A      R DSPREC2
A          OVERLAY
A          CLRL(*NO)
A          CF06(06 'Update')
A          CA07(07 'Delete')

```

```

A      DSPNUM      5 0 2 21
A                  4 2'Part description..'
A      DSPDES      25 B +1
A                  5 2'Part quantity.....'
A      DSPQTY      5 0B +1EDTCDE(1)
A                  6 2'Part price.....'
A      DSPPRC      6 2B +1EDTCDE(1)
A                  7 2'Shipment date.....'
A      DSPDAT      L B +1
A      R SFLREC1      SFL
A      DSPNUM      5 0 5 2
A      DSPDES      25      +2
A      DSPQTY      5 0      +2EDTCDE(1)
A      DSPPRC      6 2      +2EDTCDE(1)
A      DSPDAT      L      +2
A      R SFLREC2      OVERLAY
A                  SFLCTL(SFLREC1)
A                  SFLSIZ(50)
A                  SFLPAG(10)
A                  SFLDSP
A                  SFLDSPCTL
A      66          SFLCLR
A                  4 1'Number Description'
A                  +16'Quantity Price'
A                  +2'Shipment date'

```

The keyword **INDARA** defined at the file level allows you to create an indicator data structure in the program and to avoid the use of numeric indicators.

The subfile is used to display all records from the PARTS file and is populated by using the **FETCH** statement.

Based on the function key that is pressed, different SQL statements in the program are run.

6.2.6 Source code for the SQLEMBED program

You can now analyze the RPG IV program with embedded SQL statements, as shown in Example 6-17.

Example 6-17 Program SQLEMBED source

```

//*****
//  Filename SQLEMBED from DBSRC in RPGISCOOL
//  Simple ILE RPG program SQLEMBED to test embedded SQL
//
//  Examples of SELECT, INSERT, UPDATE, DELETE and FETCH
//
//  Compile this source member as program SQLEMBED using
//  command CRTSQLRPGI with COMMIT(*NONE)
//*****
//-----
// The information contained in this document has not been submitted
// to any formal IBM test and is distributed AS IS. 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. See Special Notices in redbook SG24-5402 for more.
//-----
// Display file with subfile
dcl-F DspFill WorkStn(*ext) IndDS(DispInd) SFile(SflRec1:RecNum); 1

//-----
Dcl-s RecNum      packed(3 : 0);

// Indicator data structure for display file indicators
Dcl-ds DispInd;
Exit          ind pos(3);
ListAllRec   ind pos(4);
InsertRec    ind pos(5);
UpdateRec    ind pos(6);
DeleteRec    ind pos(7);
Cancel        ind pos(12);
InvalidRec   ind pos(55);
ClearSfl     ind pos(66);
end-Ds;

// Host structure to simplify host variables in SQL statement
Dcl-ds HostStr ; 3
DspNum       packed(5 : 0);
DspDes       char(25);
DspQty       packed(5 : 0);
DspPrc       packed(6 : 2);
DspDat       date;
end-Ds;

// Indicator data structure for display file indicators
Dcl-s Action  char(1);
Dcl-c SingleRec  'S';
Dcl-c FirstRec   'F';
Dcl-c NextRec    'N';
Dcl-c EndOfFile  'E';
Dcl-c UpdRecord  'U';
Dcl-c AddRecord  'I';
Dcl-c Error      'X';

//-----
Exfmt DspRec1;
// Loop begin
Dow Not (Exit Or Cancel);
  InvalidRec = *Off;
  Select;

// Insert new record into file Parts
When InsertRec;
  Clear HostStr;
  DspNum = PartNo;
  Exfmt DspRec2;

  Exec Sql
    Insert Into Parts 4

```

```

        Values (:HostStr);

If SqlStt = '23505';
  InvalidRec = *On;
Endif;

// List all records from file Parts using subfile
When ListAllRec;
  RecNum = 0;
  ClearSfl = *On;
  Write SflRec2;
  ClearSfl = *Off;

  Exec Sql          8
    Declare C1 Cursor For
      Select * From Parts
        Order by PartNum
        For Fetch Only;

  Exec Sql          9
    Open C1;

  Exec Sql          10
    Fetch C1 Into :HostStr;

DoW SqlStt <> '02000';
  RecNum = RecNum + 1;
  Write SflRec1;

  Exec Sql          10
    Fetch C1 Into :HostStr;

EndDo;

  Exec Sql          11
    Close C1;

  Exfmt SflRec2;

// Display selected record from file Parts
Other;

  Exec Sql          7
    Select * Into :HostStr
      From Parts
      Where PartNum = :PartNo;

  If SqlStt = '02000';
    InvalidRec = *On;
  Else;
    Exfmt DspRec2;
    Select;
    When Exit Or Cancel;
      Leave;

```

```

// Update selected record in file Parts
When UpdateRec;

Exec Sql
    Update Parts
        Set PartDes = :DspDes,
            PartQty = :DspQty,
            PartPrc = :DspPrc,
            PartDat = :DspDat
    Where PartNum = :DspNum; 5

// Delete selected record from file Parts
When DeleteRec;

Exec Sql
    Delete From Parts
    Where PartNum = :DspNum; 6

EndS1;

EndIf;

EndS1;

Exfmt DspRec1;

EndDo;

// Loop end
*INLR = *On;

```

SQLIMBED program notes

The notes in this section refer to the numbers shown on the right side of Example 6-17 on page 184:

1. On the file specifications, only the display file must be defined. The database file PARTS is accessed through SQL statements and is not defined in the program.
2. On the data specifications, the indicator data structure DISPIND is used, where all display file response and conditioning indicators are defined with meaningful names. This improves the readability of our program.
3. Program variables that are used in SQL statements are called *host variables* and must be preceded by a ":" (colon). To simplify the writing of SQL statements, define the data structure on data specifications, which contains all host variables that are needed by the SQL statement. Such a data structure is known as a *host structure* and can also be externally defined.
4. To add a new record into the file, the program uses the **INSERT** statement with field values that are taken from the host structure. After inserting, check SQLSTT for the value "23505", which signals that the record with this key already exists in the file.
5. A record update is performed with the statement **UPDATE**. Assume the record was previously read. Although it is not done here, it is a preferred practice to check the return code to make sure the operation was successful.
6. A record delete is performed with statement **DELETE**. For deletes, you might not need to check the return code. If the key is not found, the row is already gone.

7. To read a single record from file **PARTS**, the program has a **SELECT** statement with an **INTO** clause to place all data into the host structure. The row is identified by a **WHERE** clause. By checking that SQLSTT equals the value '02000', you can identify that you received the 'no record found' exception.
8. To read all records from the file and put them into a subfile, use the cursor technique. The cursor should first be declared and related to the corresponding **SELECT** statement.
9. The **OPEN** statement performs a declared request and prepares the result table.
10. The **FETCH** statement, performed in the loop, reads all rows from the result table until the end of file condition (SQLSTT='02000') is reached. To keep this program simple, there is no specification of any validation of the %EOF (end of file) condition for the subfile, which can be done at this point to prevent the program from failing with a message indicating that the subfile reached its maximum capacity.
11. The **CLOSE** statement should be performed at the end of **FETCH** loop to close the cursor and prepare it for its next use.

Try it yourself: Complete the following steps:

1. Create the display file from the source:

```
CRTDSPF FILE(RPGISCOOL/DSPFIL1) SRCFILE(RPGISCOOL/DBSRC) SRCMBR(DSPFIL1)
```

2. Create the customer physical file:

```
CRTPF FILE(RPGISCOOL/PARTS) SRCFILE(RPGISCOOL/DBSRC) SRCMBR(PARTS)
```

Here is a sample of the data used to populate the file:

Part Number	Part Description	Part Quantity	Part Price	Shipment Date
12,345	Hammer	123	29.99	1999-05-01
23,456	Saw	234	20.99	1999-04-01
34,567	Hatchet	345	19.99	1999-03-01
45,678	Rasp	456	9.99	1999-02-01

3. Create the program and the program call:

```
CRTSQLRPGI OBJ(RPGISCOOL/SQLEMBED) SRCFILE(RPGISCOOL/DBSRC) COMMIT(*NONE)
CALL PGM(SQLEMBED)
```

This is a simple example to make it easier for you to understand how to use SQL in an RPG program. To make this a better program, you could separate the SQL into a service program with procedures for each of the functions. To fill the subfile, the procedure called would have returned a result set that the main program processed, rather than doing one record at a time. Then, the service program would have been available to other interfaces, such as a web process, and a 5250 emulation display. The procedures of the service program could even be defined as stored procedures and still be used by multiple interfaces.

6.3 Stored procedures

Stored procedure support is a function of DB2 SQL for IBM i. It provides a way for an SQL application to define and then start a procedure through SQL statements. Stored procedures can be used in both distributed (client/server) and non-distributed DB2 SQL for IBM i. applications.

One of the big advantages of using stored procedures is for distributed applications. The execution of one **CALL** statement on the application requester or client can perform any

amount of work on the application server. This can reduce the data transfer between the client and server and improve the performance of the distributed application.

You can define a stored procedures in two ways:

- ▶ External procedure: An external procedure can be any supported high-level language program (including ILE RPG) or a REXX procedure. The procedure does not need to contain SQL statements, but it may contain SQL statements.
- ▶ SQL procedure: An SQL procedure is defined entirely in SQL and can contain SQL statements that include SQL control statements.

You must understand the following concepts when creating and calling stored procedures:

- ▶ Stored procedure definition through the **CREATE PROCEDURE** statement.
- ▶ Stored procedure invocation through the **CALL** statement.
- ▶ Parameter passing conventions.
- ▶ Methods for returning a completion status to the program starting the procedure.

6.3.1 Creating an external procedure

To create an external procedure, use the SQL statement **CREATE PROCEDURE**, which defines the following terms:

- ▶ Procedure name
- ▶ Parameters and their attributes
- ▶ Other information about the procedure that the system uses when it calls the procedure

Consider the following example:

```
CREATE PROCEDURE mylib/procname
    (IN PARTNUM CHAR(6), INOUT PARTDES CHAR(25))
    LANGUAGE RPGLE
    MODIFIES SQL DATA
    EXTERNAL NAME mylib/progname
```

This **CREATE PROCEDURE** statement performs the following functions:

- ▶ Names the procedure and library where the procedure is stored.
- ▶ Defines two parameters as character fields. The first is input only, and the second is used for input and output.

Parameters can be defined as type **IN**, **OUT**, or **INOUT**. The parameter type determines when the values for the parameters get passed to and from the procedure.

- ▶ Indicates that the procedure is written in RPGLE. The language is important because it impacts the types of parameters that can be passed.
- ▶ Indicates the procedure is an external program that modifies SQL data.
- ▶ Names the program that is called when the procedure is invoked on a **CALL** statement.

6.3.2 Creating an SQL procedure

To create an SQL procedure, use the same SQL statement **CREATE PROCEDURE**, which defines the following items:

- ▶ Procedure name
- ▶ Parameters and their attributes
- ▶ Other information about the procedure that the system uses when it calls the procedure
- ▶ Procedure body

The procedure body is the executable part of the procedure and is a single SQL statement. If multiple SQL statements are required to accomplish the procedure logic, SQL control statements can be used to control the execution of procedure. SQL control statements consist of:

- ▶ Assignment statement
- ▶ **CALL** statement
- ▶ **CASE** statement
- ▶ Compound statement
- ▶ **FOR** statement
- ▶ **IF** statement
- ▶ **LOOP** statement
- ▶ **REPEAT** statement
- ▶ **WHILE** statement

Example 6-18 uses as input the employee number and a rating value. The procedure uses a **CASE** statement based on a rating value to determine the appropriate increase and bonus for the update.

Example 6-18 Input of employee number and rating value

```
EXEC SQL CREATE PROCEDURE UPDATE_SALARY
  (IN EMPLOYEE_NUMBER CHAR(6),
   IN RATING INT)
  LANGUAGE SQL MODIFIES SQL DATA
CASE RATING
  WHEN 1
    UPDATE mylib/EMPLOYEE
      SET SALARY = SALARY * 1.10,
          BONUS = 1000
      WHERE EMPNO = EMPLOYEE_NUMBER;
  WHEN 2
    UPDATE mylib/EMPLOYEE
      SET SALARY = SALARY * 1.05,
          BONUS = 500
      WHERE EMPNO = EMPLOYEE_NUMBER;
  ELSE
    UPDATE mylib/EMPLOYEE
      SET SALARY = SALARY * 1.03
      BONUS = 0
      WHERE EMPNO = EMPLOYEE_NUMBER;
END CASE;
```

This **CREATE PROCEDURE** statement performs the following tasks:

- ▶ Names the procedure **UPDATE_SALARY**.
- ▶ Defines the parameter **EMPLOYEE_NUMBER** as the input parameter with the character data type of length 6 and a parameter **RATING** as an input parameter with integer data type.
- ▶ Indicates that the procedure is an SQL procedure that modifies SQL data.
- ▶ Defines the procedure body. When the procedure is called, the input parameter **RATING** is checked and the appropriate update statement is run.

6.3.3 Starting a stored procedure and returning the completion status

To start a stored procedure, use the SQL **CALL** statement. This statement contains the name of the stored procedure and any arguments passed to it. Arguments may be constants, special registers, or host variables. Example 6-19 is an example of how to call a stored procedure and pass two arguments.

Example 6-19 Call a stored procedure

```
Exec Sql
Call mylib/procname (:PARTNUM, :PARTDES);
```

The easiest way to return a completion status to the SQL programs issuing the **CALL** statement is to code an extra **INOUT** type parameter and set it before returning from the procedure.

Another, more complicated method of returning a completion status is to send an escape message to the calling program (operating system program QSQCALL), which starts the procedure.

6.3.4 A stored procedure example

To illustrate the use of a stored procedure, the example used “An embedded SQL program example” on page 183 was modified. Embedded SQL statements are replaced with SQL **CALL** statements to call stored procedures. This example uses three stored procedures to show you different techniques for their creation:

- SPRCSEL** RPG IV program PROGSEL with embedded SQL statements
- SPRCUPD** RPG IV program PROGUPD with native file operations (without embedded SQL)
- SPRCDEL** SQL procedure

All these stored procedures are called from the program SPRCRUN.

Source code for program SPRCRUN

The logic of the program SPRCRUN is the same as in the program SQLEMBED shown in 6.2.6, “Source code for the SQLEMBED program” on page 184. It uses the display file DSPFIL1 to communicate with the user.

Example 6-20 shows the source code for SPRCRUN.

Note: Example 6-20 is divided into several figures to fit into this book. However, these figures should be considered one flow.

Example 6-20 Program SPRCRUN source

```
ctl-opt dftactgrp(*no);
//*****
// Source SPRCRUN from DBSRC in RPGISCOOL
// Simple ILE RPG program SPRCRUN to test stored procedures
//
// Program calls stored procedures SPRCSEL, SPRCUPD and SPRCDEL
//
// Compile this source member as program SPRCRUN (PDM Option=14)
```

```

// or use command CRTSQLRPGI with COMMIT(*NONE) and DATFMT(*ISO)
//*****
//-----
// The information contained in this document has not been submitted
// to any formal IBM test and is distributed AS IS. 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. See Special Notices in redbook SG24-5402 for more.
//-----
// Display file with subfile
dcl-F DspFill WorkStn(*ext) IndDS(DispInd) SFile(SflRec1:RecNum); 1
//-----
Dcl-s RecNum     packed(3 : 0);

// Copy book for prototypes (not needed here) and actions
/copy dbsrc,progpartpr

// Indicator data structure for display file indicators
Dcl-ds DispInd; 2
Exit      ind pos(3);
ListAllRec ind pos(4);
InsertRec  ind pos(5);
UpdateRec  ind pos(6);
DeleteRec  ind pos(7);
Cancel     ind pos(12);
InvalidRec ind pos(55);
ClearSfl   ind pos(66);
end-Ds;

// Action parameter. See copybook for meanings
Dcl-s Action  char(1); 3

//-----
// Stored Procedures SPrcSel written with embedded SQL
//
exec sql
    set option datfmt = *iso; 4

Exec Sql
    Create or replace Procedure SprcSel
        (InOut Action Char(1), InOut PartNum Numeric(5 , 0),
         Out PartDes Char(25), Out PartQty Numeric(5 , 0),
         Out PartPrc Numeric(6 , 2), Out PartDat Date)
        Language RPGLE
        Modifies SQL Data
        Program type sub
        External Name Progparts(progSel);

// Stored Procedures SPrcUpd written with native file operations
//
Exec Sql
    Create or replace Procedure SprcUpd
        (InOut Action Char(1), In PartNum Numeric(5 , 0),
         In PartDes Char(25), In PartQty Numeric(5 , 0),

```

```

        In PartPrc Numeric(6 , 2), In PartDat Date
        Language RPGLE
        Modifies SQL Data
        Program type sub
        External Name progparts(ProgUpd);

// Stored Procedures SPrcDel written in SQL
Exec Sql
    Create or replace Procedure SprcDel
        (In PartNo Numeric(5 , 0))
        Language SQL
        Modifies SQL Data
        Delete From Parts Where PartNum = PartNo; 7

Exfmt DspRec1;
// Loop begin
Dow Not (Exit Or Cancel);
    InvalidRec = *Off;
    Select;

// Insert new record into file Parts
When InsertRec;
    Clear DspDes;
    Clear DspQty;
    Clear DspPrc;
    Clear DspDat;
    DspNum = PartNo;
    Exfmt DspRec2;
    Action = AddRecord;

Exec Sql
    Call SprcUpd 8
        (:Action, :PartNo, :DspDes, :DspQty, :DspPrc, :DspDat);

If Action = Error;
    InvalidRec = *On;
Endif;

// List all records from file Parts using subfile
When ListAllRec;
    RecNum = 0;
    ClearSfl = *On;
    Write SflRec2;
    ClearSfl = *Off;
    Action = FirstRec;

Exec Sql
    Call SprcSel 12
        (:Action, :DspNum, :DspDes, :DspQty, :DspPrc, :DspDat);

Dow Action <> EndOfFile;
    RecNum = RecNum + 1;
    Write SflRec1;
    Action = NextRec;

```

```

      Exec Sql
      Call SprcSel
      (:Action, :DspNum, :DspDes, :DspQty, :DspPrc, :DspDat); 13
      EndDo;

      Exfmt SflRec2;

// Display selected record from file Parts
Other;
Action = SingleRec;
DspNum = PartNo;

Exec Sql
Call SprcSel 11
(:Action, :DspNum, :DspDes, :DspQty, :DspPrc, :DspDat);

If Action = Error;
  InvalidRec = *On;
Else;
  Exfmt DspRec2;
  Select;
  When Exit Or Cancel;
    Leave;

// Update selected record in file Parts
When UpdateRec;
Action = UpdRecord;

Exec Sql
Call SprcUpd 9
(:Action, :PartNo, :DspDes, :DspQty, :DspPrc, :DspDat);

When DeleteRec;
  Exec Sql
  Call SPrcDel (:PartNo); 10

EndSI;

EndIf;

EndSI;

Exfmt DspRec1;
EndDo;

// Loop end
*INLR = *On;

```

SPRCRUN program notes

The notes in this section refer to the numbers shown on the right side of Example 6-20 on page 191:

1. On the file specifications, only the display file must be defined. The database file PARTS is accessed through stored procedures and is not defined in the program.

2. On the data specifications, this example uses the indicator data structure DISPIND, where all display file response and conditioning indicators are defined with meaningful names.
3. The parameter **ACTION** is used for communication with stored procedures. Defined constants provide meaningful names for all of its values to improve the readability of the program. The actions are defined in a common copy member so all the programs using the procedures have access to the same list.
4. The **SET OPTION** SQL statement is used so that the date fields are defined as ISO standard dates. Without this option, the **CREATE PROCEDURE** statements define the date parameters to be whatever the date format of the job running the program is.
5. The stored procedure SPRCSEL should be created before it is used. It requires two input/output parameters and four output parameters, and relates to the RPG IV procedure PROGSEL in service program PROGPARTS. As you see later, this procedure contains the embedded SQL statements **SELECT**, **FETCH**, **OPEN**, and **CLOSE** cursor to read data from the PARTS file.
6. Another stored procedure, SPRCUPD, is created. It requires one input/output parameter and five input parameters, and relates to the RPG IV procedure PROGUPD, also found in the PROGPARTS service program. This procedure contains native file operations **CHAIN**, **WRITE**, and **UPDATE** to show you that any procedure or program can be declared and used as a stored procedure.
7. The stored procedure SPRCDEL is created. It has only one input parameter and is written as a single SQL statement.

All these stored procedures can be created outside of the program by using interactive SQL (STRSQL) or any other SQL interface. Created stored procedures are cataloged in the SQL catalog, and can be used from any program using the SQL **CALL** statement. Notice that all of them use the **CREATE OR REPLACE PROCEDURE** statement, so you can run this program multiple times without getting a 'procedure already exists' SQL error.

8. Stored procedure SPRCUPD is called to insert a record.
9. Stored procedure SPRCUPD is called to update a record.
10. Stored procedure SPRCDEL is called to delete a record.
11. Stored procedure SPRCSEL is called to read a single record.
12. Stored procedure SPRCSEL is called to read the first record.
13. Stored procedure SPRCSEL is called to read the next record.

Source code for PROGPARTS service program

This service program contains the procedures SPRCSEL and SPRCUPD. Typically, you want to have a service program that contains all the procedures that process information from a particular table or set of tables, which gives you more flexibility when it comes to changing a table. You can reuse the code. When you define service programs, you want to externalize the prototypes of the procedures as well.

Example 6-21 shows is the prototype source member for the PROGPARTS service program.

Example 6-21 Source code for PROGPARTS service program

```
/if defined(progpartpr_h)
/eof
/else
/define progpartpr_h
//*****
//  Source PROGPARTPR from DBSRC in RPGISCOOL
//  Prototypes for RPG service program PROGPARTS with two procedures:
```

```

//  progSel - example of embedded SQL with SELECT and FETCH
//  progUpd - example of native I/O used as stored procedure
//
//*****=====
// Action parameter and return values          1
//
Dcl-c SingleRec      'S';
Dcl-c FirstRec       'F';
Dcl-c NextRec        'N';
Dcl-c UpdRecord      'U';
Dcl-c AddRecord      'I';
Dcl-c EndOfFile      'E';
Dcl-c Error          'X';

// Prototype and entry parameter definition for progUpd          2
Dcl-pr ProgUpd;
  Action  char(1);
  PartNo zoned(5 : 0);
  PartDs char(25);
  PartQy zoned(5 : 0);
  PartPr zoned(6 : 2);
  PartDt date;
end-pr;

// Prototype and entry parameter definition for progSel          3
Dcl-pr ProgSel;
  Action  char(1);
  PartNo zoned(5 : 0);
  PartDs char(25);
  PartQy zoned(5 : 0);
  PartPr zoned(6 : 2);
  PartDt date;
end-pr;
/endif

```

Bridgeport notes

The notes in this section refer to the numbers shown on the right side of Example 6-21 on page 195:

1. The allowed values of the **ACTION** parameter are given here. Now, everyone that uses these prototypes can use the same constants for setting the action and checking the return.
2. The prototype for the preadapt procedure with its parameters. As all but the first parameter is used as input only, you might have used the **CONST** keyword with them. This example uses the procedures as SQL procedures, so the prototypes are not needed by the calling program. Having the prototypes means that you can use these procedures as either an SQL procedure or as an RPG procedure.
3. The Pergolas procedure with its parameters. These are used as output only by the SQL procedure.

Example 6-22 through shows the code for the PROGPARTS service program.

Example 6-22 Source for PROGPARTS service program

```

ctl-opt nomain; 1
//*****
//  Source PROGPARTS from DBSRC in RPGISCOOL
//  RPG service program PROGPARTS with two procedures:
//  progSel - example of embedded SQL with SELECT and FETCH
//  progUpd - example of native I/O used as stored procedure
//
//  Compile this source member as service program PROGPARTS
//  use command CRTSQLRPGI with COMMIT(*NONE)
//*****
//-----
// The information contained in this document has not been submitted
// to any formal IBM test and is distributed AS IS. 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. See Special Notices in redbook SG24-5402 for more.
//-----

/copy dbsrc,progpartpr 2

//*****
// Procedure progSel
// Uses SQL to select a record from the parts table.
//*****
dcl-proc progSel export; 3
dcl-pi ProgSel;
  Action  char(1);
  PartNo zoned(5 : 0);
  PartDs  char(25);
  PartQy  zoned(5 : 0);
  PartPr  zoned(6 : 2);
  PartDt  date;
end-Pi;

//-----
Select;
When Action = SingleRec;

// Read single record from file Parts
Exec Sql 4
  Select * Into :PartNo, :PartDs, :PartQy, :PartPr, :PartDt
    From Parts
    Where PartNum = :PartNo;

If SqlStt = '02000';
  Action = Error;
Endif;

When Action = FirstRec;

```

```

Exec Sql          5
  Declare C1 Cursor For
    Select * From Parts
      Order by PartNum
      For Fetch Only;

Exec Sql          6
  Open C1;

// Read first record from file Parts
Exec Sql
  Fetch C1 Into :PartNo, :PartDs, :PartQy, :PartPr, :PartDt; 7

When Action = NextRec;

// Read next record from file Parts
Exec Sql
  Fetch C1 Into :PartNo, :PartDs, :PartQy, :PartPr, :PartDt; 7

If SqlStt = '02000';
  Action = EndOfFile;

  Exec Sql          8
    Close C1;

Endif;

EndSI;

Return;

end-proc progSel;

//*****
// Procedure progUpd
// Native I/O used to update or write a row into the parts table
//*****
dcl-proc Progupd export;          9
dcl-pi ProgUpd;
  Action  char(1);
  PartNo zoned(5 : 0);
  PartDs char(25);
  PartQy zoned(5 : 0);
  PartPr zoned(6 : 2);
  PartDt date;
end-pi;

dcl-f Parts Disk(*ext) usage(*update: *output) Keyed qualified; 10

dcl-ds partsDS likeref(PARTS.partr);           11

//-----
// Read a record from file Parts
Chain PartNo parts partsDS;          12
partsDS.PartNum = PartNo;

```

```

partsDS.PartDes = PartDs;
partsDS.PartQty = PartQty;
partsDS.PartPrc = PartPr;
partsDS.PartDat = PartDt;

Select;
When Action = UpdRecord And %FOUND(Parts);          13
  // Update a record in file Parts
  Update parts.partr partsDS;                      14
When Action = AddRecord And Not %FOUND(Parts);
  // Write a record into file Parts
  Write parts.partr partsDS;                      15
Other;
  // Return error in other cases
  Action = Error;
EndSL;

Return;

end-proc progUpd;

```

PROGPARTS service program notes

The notes in this section refer to the numbers shown on the right side of Example 6-22 on page 197:

1. The control option specification (old H specification) used 'naming' to tell the compiler there is no mainline procedure in this source member.
2. The copy statement brings in the procedure prototypes and the **ACTION** parameter values.
3. The PROGSEL procedure starts here. It is defined as being exportable, so it is accessible to outside callers.
4. To read a single record, use the **SELECT** statement with the **INTO** clause. If the record does not exist, the **ACTION** parameter is returned with the value "X" to signal an error.
5. When reading all records from the file, the cursor technique is required. The **DECLARE** statement defines the cursor and relates it to the corresponding **SELECT** statement.
6. The **OPEN** statement runs the defined **SELECT** and prepares the result table.
7. The **FETCH** statement reads a single record from the result table. At the end of the result table, when SQLSTT='02000', the **ACTION** parameter is returned with the value "E".
8. After reading all records from the result table, the cursor should be closed to be ready for the next open.
9. The PROGUPD procedure starts here. It is defined as exportable. The update procedure must use native I/O instead of SQL. At some future date, you want to convert it to SQL. You can do that without needing to recompile any of the programs that use the procedures in this service program.
10. You can now use RPG to define file specifications inside procedures. In this example, the file is qualified so that when RPG uses anything from the file, it must have the file name to identify it.
11. When you define a file specification inside a procedure, you must define a data structure that is based on the file, which means RPG can keep the fields straight in case you do this task in multiple procedures. Notice how the record format, PARTR, is qualified with the file name.

12. With the CHAIN operation, you are checking whether the record exists and retrieving the current values of the record. The record is put in the parts data structure where it is replaced by the values from the parameters.
13. You check the WHEN statement to discover whether the action is requested and whether a record was found.
14. UPDATE updates the last record read (in this case, with the CHAIN). Notice how the record format name is qualified and uses the data structure as the input for the values.
15. WRITE is similar to the update, but it is used only if you did not find a record on the CHAIN.

Try it yourself: You can try this example by compiling the code from this section on your IBM i system. To create the programs, run the following commands:

```
CRTSQLRPGI OBJ(RPGISCOOL/SPRCRUN) SRCFILE(RPGISCOOL/DBSRC) +
COMMIT(*NONE) DATFMT(*ISO)
CRTSQLRPGI OBJ(RPGISCOOL/PROGPARTS) SRCFILE(RPGISCOOL/DBSRC) COMMIT(*NONE)
OBJTYPE(*MODULE)
CRTSRVPGM SRVPGM(RPGISCOOL/PROGPARTS) EXPORT(*ALL)
```

To run the program, run the following command:

```
CALL PGM(SPRCRUN)
```

6.4 DB2 call level interface

The DB2 call level interface (DB2 CLI) is a callable SQL programming interface that is supported in most DB2 environments. A callable SQL interface is an application program interface (API) for database access that uses function calls to invoke dynamic SQL statements.

It is an alternative to embedded dynamic SQL. The important difference between embedded dynamic SQL and DB2 CLI lies in how the SQL statements are invoked. On the IBM i, this interface is available to any of the ILE languages, including RPG IV.

DB2 CLI also provides full Level 1 Microsoft Open Database Connectivity (ODBC) support, plus many Level 2 functions. ODBC is based on the emerging X/Open and SQL Access Group Call Level Interface specification.

The X/Open company and the SQL Access Group (SAG) are jointly developing a standard specification for a callable SQL interface that is referred to as X/Open CLI or SAG CLI. The goal of this interface is to increase the portability of applications by enabling them to become independent of any one database server.

Microsoft developed a callable SQL interface called Open Database Connectivity (ODBC) for MS Windows based on a preliminary draft of X/Open CLI. ODBC has expanded X/Open CLI and provides extended functions supporting additional capability. ODBC provides a Driver Manager for Windows, which offers a central point of control for each ODBC.

6.4.1 Differences between DB2 CLI and embedded SQL

An application that uses an embedded SQL interface requires a precompiler to convert the SQL statements into code, which is then compiled, bound to the database, and run. In contrast, a DB2 CLI application does not require precompilation or binding, but instead uses a standard set of functions to run SQL statements and related services at run time.

This difference is important because traditionally precompilers have been specific to a database product, which effectively ties your applications to that product. DB2 CLI enables you to write portable applications that are independent of any particular database product. This independence means a DB2 CLI application does not have to be recompiled or rebound to access different database products, but rather selects the appropriate one at run time.

DB2 CLI can run any SQL statement that can be prepared dynamically in embedded SQL. This is ensured because DB2 CLI does not actually run the SQL statement itself, but passes it to the DBMS for dynamic execution.

Advantages of using DB2 CLI

The DB2 CLI interface has several key advantages over embedded SQL:

- ▶ It is ideally suited for a client-server environment, in which the target database is not known when the application is built. It provides a consistent interface for running SQL statements, regardless of the database server to which the application is connected.
- ▶ It increases the portability of applications by removing the dependence on precompilers. Applications are distributed not as source code that must be preprocessed for each database product, but as compiled applications or runtime libraries.
- ▶ DB2 CLI applications do not have to be bound to each database to which they connect.
- ▶ DB2 CLI applications can connect to multiple databases simultaneously.
- ▶ DB2 CLI applications are not responsible for controlling global data areas, such as SQLCA and SQLDA because they are with embedded SQL applications. Instead, DB2 CLI allocates and controls the necessary data structures and provides a handle for the application to reference them.
- ▶ DB2 CLI support is included in the base operating system for IBM i and is available as a part of system APIs.

Deciding which interface to use

The interface you choose depends on your application.

DB2 CLI is ideally suited for query-based applications that require portability and do not require the APIs or utilities that are offered by a particular DBMS (for example, catalog database, backup, and restore). This does not mean that DBMS-specific APIs cannot be called from an application by using DB2 CLI, but rather that the application no longer is as portable.

Another important consideration is the performance comparison between a dynamic and static SQL. Dynamic SQL is prepared at run time, while static SQL is prepared during the precompile stage. Because preparing statements requires additional processing time, static SQL might be more efficient. If you choose static over dynamic SQL, then DB2 CLI is not an option.

In most cases, the choice between either interface is open to personal preference. Your previous experience might make one alternative seem more intuitive than the other.

6.4.2 Writing a DB2 CLI application

This section introduces a conceptual view of a typical DB2 CLI application. A DB2 CLI application can be broken down into a set of tasks. Some of these tasks are organized into discrete steps, and others might apply throughout the application. Each task is carried out by calling one or more DB2 CLI functions. Every DB2 CLI application contains the three main tasks that are shown in Figure 6-2.

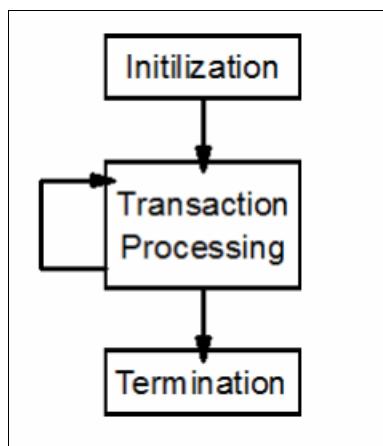


Figure 6-2 Conceptual view of a DB2 CLI application

The functions must be called in the sequence shown or an error is returned. The tasks of the application are explained here:

Initialization

This task allocates and initializes some resources in preparation for the main Transaction Processing task.

Transaction Processing

This is the main task of the application. SQL statements are passed to DB2 CLI to query and modify the data.

Termination

This task frees allocated resources. The resources generally consist of data areas that are identified by unique handles. After the resources are freed, these handles can be used by other tasks.

In addition to the three tasks listed here, there are general tasks, such as handling diagnostic messages, which occur throughout an application.

6.4.3 Initialization and termination

Figure 6-3 on page 203 shows the function call sequences for both the initialization and termination tasks. The transaction processing task in the middle of the diagram is shown in Figure 6-4 on page 204.

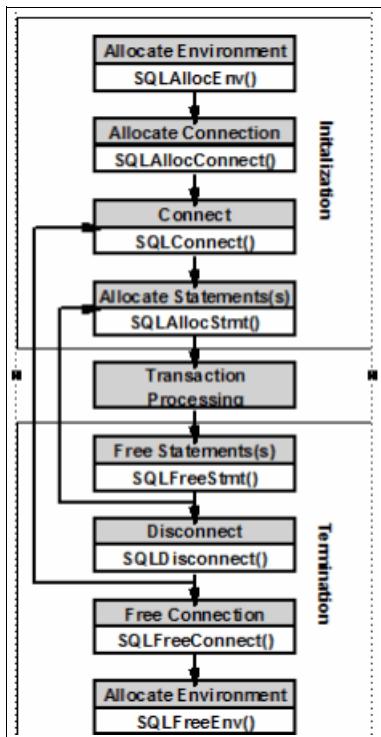


Figure 6-3 Conceptual view of initialization and termination tasks

The initialization task allocates and initializes environment and connection handles. The termination task frees them.

A handle is a variable that refers to a data object that is controlled by DB2 CLI. Using handles frees the application from having to allocate and manage global variables or data structures, such as the SQLDA or SQLCA, which are used in embedded SQL interfaces for IBM DBMS. An application then passes the appropriate handle when it calls other DB2 CLI functions.

There are three types of handles:

Environment handle The environment handle refers to the data object that contains global information regarding the state of the application. This handle is allocated by calling `SQLAllocEnv()` and freed by calling `SQLFreeEnv()`. An environment handle must be allocated before a connection handle can be allocated. Only one environment handle can be allocated per application.

Connection handle A connection handle refers to a data object that contains information that is associated with a connection managed by DB2 CLI. This information includes general status information, transaction status, and diagnostic information. Each connection handle is allocated by calling `SQLAllocConnect()` and freed by calling `SQLFreeConnect()`. An application must allocate a connection handle for each connection to a database server.

Statement handle	A statement handle refers to the data object that contains information about an SQL statement that is managed by DB2 CLI. This information includes information such as dynamic arguments, cursor information, bindings for dynamic arguments and columns, result values, and status information. Each statement handle is allocated by calling <code>SQLAllocStmt()</code> and freed by calling <code>SQLFreeStmt()</code> , and must be associated with a connection handle.
-------------------------	--

6.4.4 Transaction processing

Figure 6-4 shows the steps and the DB2 CLI functions in the transaction processing task. This task involves five steps:

1. Allocate statement handles.
2. Prepare and run SQL statements.
3. Process the results.
4. Free the statement handles.
5. Commit or rollback.

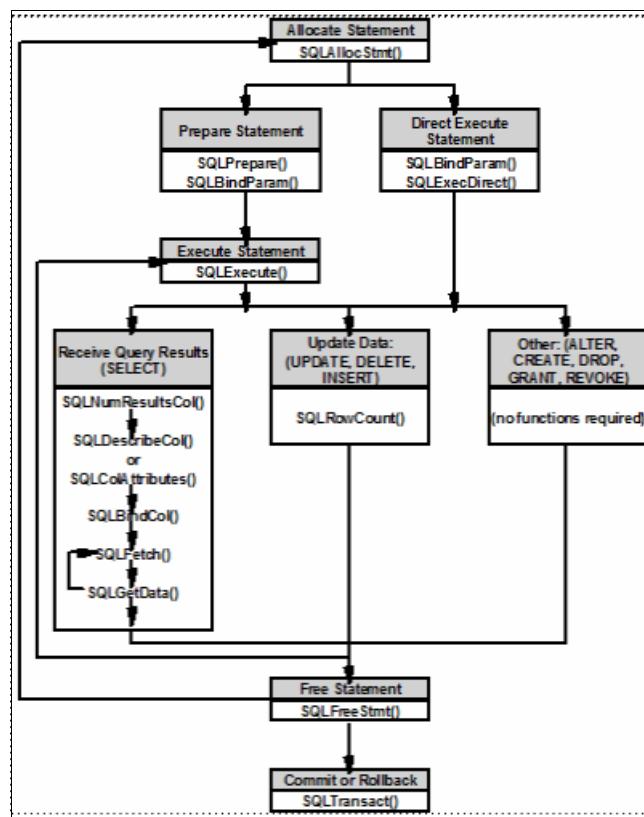


Figure 6-4 Transaction processing

Allocating statement handles

To run a statement, allocate a statement handle by using `SQLAllocStmt()`. Each statement handle must be associated with a connection handle.

Preparation and running of SQL statements

After a statement handle is allocated, there are two methods of specifying and running SQL statements:

- ▶ Prepare and then run in order:
 - a. Call **SQLPrepare()** with an SQL statement as an argument.
 - b. Call **SQLBindParameter()**, if the SQL statement contains parameter markers.
 - c. Call **SQLExecute()**.
- ▶ Run directly:
 - a. Call **SQLBindParameter()** if the SQL statement contains parameter markers.
 - b. Call **SQLExecDirect()** with an SQL statement as an argument.

The ‘prepare and run in order’ method splits the preparation of the statement from the execution. This method is used in the following situations:

- ▶ The statement is run repeatedly (usually with different parameter values), which means that you do not have to prepare the same statement more than once.
- ▶ The application requires information about the columns in the result set, before statement execution.

The ‘run directly’ method combines the preparation step and the execution step into one. This method is used in the following situations:

- ▶ The statement is run once, which means that you do not have to call two functions to run the statement.
- ▶ The application does not require information about the columns in the result set before the statement is run.

Both methods can use parameter markers in place of an expression (or a host variable in embedded SQL) in an SQL statement.

Parameter markers are represented by the "?" (question mark) character and indicate the position in the SQL statement where the contents of application variables are substituted when the statement is run. The markers are referenced sequentially, from left to right, starting at 1.

When an application variable is associated with a parameter marker, it is bound to the parameter marker. Binding is carried out by calling the **SQLBindParam()** function, which requires the following information:

- ▶ The number of the parameter marker
- ▶ A pointer to the application variable
- ▶ The SQL type of the parameter
- ▶ The data type and length of the variable

The application variable is called a *deferred argument* because only the pointer is passed when **SQLBindParameter()** is called. No data is read from the variable until the statement is run, which applies to both buffer arguments and arguments that indicate the length of the data in the buffer. You can use deferred arguments with the application to modify the contents of the bound parameter variables and repeat the execution of the statement with the new values.

In *DB2 for IBM i SQL Call Level Interface*, SC41-5806, for Version 7 Release 1 and Version 7 Release 2, you can find the statement that the **SQLBindParameter()** function allows you to bind a variable of a different type from that required by the SQL statement. In this case, DB2

CLI converts the contents of the bound variable to the correct type. The types must be compatible or an error is returned.

Processing results

The next step after the statement is run depends on the type of SQL statement.

Processing SELECT statements

When processing a **SELECT** statement, the following steps are generally needed to retrieve each row of the result set:

1. Establish the structure of the result set, number of columns, column types, and lengths.
2. Optionally, bind application variables to columns to receive the data.
3. Repeatedly fetch the next row of data, and receive it into the bound application variables.
4. Optionally, columns that were not previously bound can be retrieved by calling **SQLGetData()** after each successful fetch.

Each of these steps requires some diagnostic checks:

- ▶ The first step requires you to analyze the executed or prepared statement. If the SQL statement was generated by the application, this step is not necessary because the application knows the structure of the result set and the data types of each column. If the SQL statement was generated at run time (entered by a user), the application must query the following items:
 - The number of columns
 - The type of each column
 - The names of each column in the result set

This information can be obtained by calling **SQLNumResultCols()** and **SQLDescribeCol()** (or **SQLColAttribute()**) after preparing the statement or after running the statement.

- ▶ The second step allows the application to retrieve column data directly into an application variable on the next call to **SQLFetch()**. For each column to be retrieved, the application calls **SQLBindCol()** to bind an application variable to a column in the result set. Similar to variables bound to parameter markers that use **SQLBindParameter()**, columns are bound by using deferred arguments. This time, the variables are output arguments, and data is written to them when **SQLFetch()** is called. **SQLGetData()** also can be used to retrieve data, so calling **SQLBindCol()** is optional.
- ▶ The third step is to call **SQLFetch()** to fetch the first or next row of the result set. If any columns are bound, the application variable is updated. If any data conversion was indicated by the data types specified on the call to **SQLBindCol()**, the conversion occurs when **SQLFetch()** is called.
- ▶ The last (optional) step is to call **SQLGetData()** to retrieve any columns that were not previously bound. All columns can be retrieved this way, provided they were not bound, or a combination of both methods can be used. **SQLGetData()** is also useful for retrieving variable length columns in smaller pieces, which cannot be done with bound columns. Data conversion can also be indicated here, as in **SQLBindCol()**.

Processing UPDATE, DELETE, and INSERT statements

If the statement is modifying data (**UPDATE**, **DELETE**, or **INSERT**), no action is required, other than the normal check for diagnostic messages. In this case, **SQLRowCount()** can be used to obtain the number of rows affected by the SQL statement.

Freeing statement handles

To end processing for a particular statement handle, call `SQLFreeStmt()`. This function can be used to do one or more of the following actions:

- ▶ Unbind all columns.
- ▶ Unbind all parameters.
- ▶ Close any cursors and discard the results.
- ▶ Drop the statement handle and release all associated resources.

The statement handle can be reused provided that it is not dropped.

Committing or rolling back

The last step is to either commit or roll back the transaction by using `SQLTransact()`. A transaction is a recoverable unit of work or a group of SQL statements that can be treated as one atomic operation, which means that all the operations within the group will be completed (committed) or undone (rolled back) as though they were a single operation.

When using DB2 CLI, transactions are started implicitly with the first access to the database by using `SQLPrepare()`, `SQLExecDirect()`, or `SQLGetTypeInfo()`. The transaction ends when you use `SQLTransact()` to either roll back or commit the transaction, which means that any SQL statements that are run between them are treated as one unit of work.

6.4.5 Diagnostic tests

Diagnostic tests refers to dealing with warning or error conditions that are generated within an application. There are two levels of diagnostic tests when calling DB2 CLI functions:

- ▶ Return Codes
- ▶ SQLSTATEs (diagnostic messages)

Return codes

Each function gives a return code to inform the program about possible errors or exceptions. Table 6-1 describes all DB2 CLI function return codes.

Table 6-1 DB2 CLI function return codes

Return code	Value	Description
SQL_SUCCESS	0	The function completed successfully. No additional SQLSTATE information is available.
SQL_SUCCESS_WITH_INFO	1	The function completed successfully with a warning or other information. Call <code>SQLError()</code> to receive the SQLSTATE and other error information.
SQL_NO_DATA_FOUND	100	The function returned successfully, but no relevant information was found.
SQL_ERROR	-1	The function failed. Call <code>SQLError()</code> to receive the SQLSTATE and any other error information.
SQL_INVALID_HANDLE	-2	The function failed due to an invalid handle (environment, connection, or statement handle) passed as an input argument.

SQLSTATEs

SQLSTATEs are alphanumeric strings of five characters (bytes) with a format of *ccsss*, where *cc* indicates a class and *sss* indicates a subclass.

An SQLSTATE may have the following classes:

- ▶ 01 indicates a warning.
- ▶ HY is generated by the CLI driver (either DB2 CLI or ODBC)

Follow these guidelines for using SQLSTATEs within your application:

- ▶ Always check the function return code before calling `SQLGetDiagRec()` to determine whether diagnostic information is available.
- ▶ Use SQLSTATEs rather than the native error code.
- ▶ To increase your application's portability, build dependencies only on the subset of DB2 CLI SQLSTATEs that are defined by the X/Open specification, and return the additional ones as information only.
- ▶ For maximum diagnostic information, return the text message along with the SQLSTATE (if applicable, the text message includes the IBM defined SQLSTATE). It is also useful for the application to print out the name of the function that returned the error.

6.4.6 Data types and data conversion

When writing a DB2 CLI application, it is necessary to work with both SQL data types and RPG IV data types. This situation is unavoidable because DBMS uses SQL data types and the application must use RPG IV data types. The application must match RPG IV data types to SQL data types when transferring data between DBMS and the application (when calling DB2 CLI functions).

To help address this, DB2 CLI provides symbolic names for the various data types, and manages the transfer of data between the DBMS and the application. It also performs data conversion if required. To accomplish this task, DB2 CLI must know both the source and target data type, which requires the application to identify both data types by using symbolic names.

The symbolic names are used in functions `SQLBindParam()`, `SQLBindCol()`, and `SQLGetData()` to indicate the data types of the arguments.

SQL symbolic names are defined as integer values and should be declared in an include file to be available to all applications.

The code snippet that is shown in Example 6-23 illustrates how symbolic names can be defined.

Example 6-23 Symbolic names code

```

//*****
// Standard SQL data types
//*****
Dcl-c SQL_CHAR      1;
Dcl-c SQL_NUMER     2;
Dcl-c SQL_DECIM    3;
Dcl-c SQL_INTEG    4;
Dcl-c SQL_SMINT    5;
Dcl-c SQL_FLOAT     6;
Dcl-c SQL_REAL      7;

```

```

Dcl-c SQL_DOUBLE      8;
Dcl-c SQL_DATTIM      9;
Dcl-c SQL_VARCH       12;
Dcl-c SQL_BLOB        13;
Dcl-c SQL_CLOB        14;
Dcl-c SQL_DBCLOB      15;
Dcl-c SQL_DATALINK    16;
Dcl-c SQL_WCHAR        17;
Dcl-c SQL_WVARCHAR     18;
Dcl-c SQL_BIGINT       19;
Dcl-c SQL_BLOB_LOCATOR 20;
Dcl-c SQL_CLOB_LOCATOR 21;
Dcl-c SQL_DBCLOB_LOC   22;
Dcl-c SQL_UTF8_CHAR    23;
Dcl-c SQL_GRAPHIC      95;
Dcl-c SQL_VARGRAPHIC   96;
Dcl-c SQL_BINARY        -2;
Dcl-c SQL_VARBINARY     -3;
Dcl-c SQL_DATE          91;
Dcl-c SQL_TIME          92;
Dcl-c SQL_TIMESTAMP     93;
Dcl-c SQL_CODE_DATE     1;
Dcl-c SQL_CODE_TIME     2;
Dcl-c SQL_CODD_TIMESTAMP 3;
Dcl-c SQL_ALLTYPES      0;
Dcl-c SQL_DECFLOAT     -360;
Dcl-c SQL_XML           -370;

```

6.4.7 Functions

All DB2 CLI functions are available as procedures in the service program QSQCLI in the library QSYS. In IBM i version7, Release 2, there are 81 functions. They are described in the *Database SQL Call Level Interface*:

https://www.ibm.com/support/knowledgecenter/en/ssw_ibm_i_73/cli/rzadpkickoff.htm

Note: Only the descriptions of those functions that are needed in this example are included in this section.

SQLAllocEnv(): Allocating an environment handle

SQLAllocEnv() allocates an environment handle and its associated resources. An application must call this function before **SQLAllocConnect()** or any other DB2 CLI functions. The **henv** value is passed in all later function calls that require an environment handle as input.

Here is the C syntax:

```
SQLRETURN SQLAllocEnv (SQLHENV      *phenv);
```

Table 6-2 shows the parameters for the **SQLAllocEnv()** function.

Table 6-2 Parameters for the **SQLAllocEnv()** function

Argument	Description	Use	RPG data type	C data type
phenv	Pointer to the environment handle	Output	Integer(10)	SQLHENV *

There can be only one active environment at any one time per application. Any later calls to **SQLAllocEnv()** return the existing environment handle. The return codes are SQL_SUCCESS and SQL_ERROR.

SQLAllocConnect(): Allocating an connection handle

SQLAllocConnect() allocates a connection handle and its associated resources within the environment that is identified by the input environment handle. **SQLAllocEnv()** must be called before calling this function.

Here is the C syntax:

```
SQLRETURN SQLAllocConnect (SQLHENV    henv,
                           SQLHDBC   *phdbc);
```

Table 6-3 shows the parameters for the **SQLAllocConnect()** function.

Table 6-3 Parameters for the SQLAllocConnect() function

Argument	Description	Use	RPG data type	C data type
henv	Environment handle	Input	Integer(10)	SQLHENV
phdbc	Pointer to the connection handle	Output	Integer(10)	SQLHDBC *

The output connection handle is used by DB2 CLI to reference all information related to the connection, including general status information, the transaction state, and error information. The return codes are SQL_SUCCESS, SQL_ERROR, and SQL_INVALID_HANDLE.

SQLConnect(): Connecting to a data source

SQLConnect() establishes a connection to the target database. The application must supply a target SQL database, and optionally an authorization-name and an authentication-string.

SQLAllocConnect() must be called before calling this function. This function must be called before calling **SQLAllocStmt()**.

Here is the C syntax:

```
SQLRETURN SQLConnect (SQLHDBC      hdbc,
                      SQLCHAR     *szDSN,
                      SQLSMALLINT cbDSN,
                      SQLCHAR     *szUID,
                      SQLSMALLINT cbUID,
                      SQLCHAR     *szAuthStr,
                      SQLSMALLINT cbAuthStr);
```

Table 6-4 shows the parameters for the **SQLConnect()** function.

Table 6-4 Parameters for the SQLConnect() function

Argument	Description	Use	RPG data type	C data type
hdbc	Connection handle	Input	Integer(10)	SQLHDBC
*szDSN	Database name	Input	Pointer	SQLCHAR *
cbDSN	Length of the database name	Input	Integer(5)	SQLSMALLINT
*szUID	User ID	Input	Pointer	SQLCHAR *
cbUID	Length of the user ID	Input	Integer(5)	SQLSMALLINT

Argument	Description	Use	RPG data type	C data type
*szAuthStr	User password	Input	Pointer	SQLCHAR *
cbAuthStr	Length of the password	Input	Integer(5)	SQLSMALLINT

You can define various connection characteristics (options) in the application by using **SQLSetConnectOption()**.

The database name must already be defined on the system for the connect to work. On an IBM i system, you can use the Work with Relational Database Directory Entries (**WRKRDBDIRE**) command to determine which data sources already are defined and to define optionally additional data sources.

The return codes are SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, and SQL_INVALID_HANDLE.

SQLSetConnectOption(): Setting a connection option

SQLSetConnectOption() sets connection attributes for a particular connection.

Here is the C syntax:

```
SQLRETURN SQLSetConnectOption (HDBC      hdbc,
                               SQLSMALLINT fOption,
                               SQLPOINTER vParam);
```

Table 6-5 shows the parameters for the **SQLSetConnectOption()** function.

Table 6-5 Parameters for the SQLSetConnectOption() function

Argument	Description	Use	RPG data type	C data type
hdbc	Connection handle	Input	Integer(10)	HDBC
foption	Connect option to set	Input	Integer(5)	SQLSMALLINT
vParam	Value that is associated with the option.	Input	Pointer	SQLPOINTER

SQLSetConnectOption() provides the same function as **SQLSetConnectAttr()**. Both functions are supported for compatibility reasons.

All connection and statement options that are set through **SQLSetConnectOption()** persist until **SQLFreeConnect()** is called or the next **SQLSetConnectOption()** call.

The return codes are SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, and SQL_INVALID_HANDLE.

SQLAllocStmt(): Allocating a statement handle

SQLAllocStmt() allocates a new statement handle and associates it with the connection that is specified by the connection handle. There is no defined limit on the number of statement handles that can be allocated at any one time. **SQLConnect()** must be called before calling this function. This function must be called before **SQLBindParam()**, **SQLPrepare()**, **SQLExecute()**, **SQLExecDirect()**, or any other function that has a statement handle as one of its input arguments.

Here is the C syntax:

```
SQLRETURN SQLAllocStmt (SQLHDBC      hdbc,
                        SQLHSTMT    *phstmt);
```

Table 6-6 shows the parameters for the **SQLAllocStmt()** function.

Table 6-6 Parameters for the SQLAllocStmt() function

Argument	Description	Use	RPG data type	C data type
hdbc	Connection handle	Input	Integer(10)	SQLHDBC
*phstmt	Pointer to statement handle	Output	Integer(10)	SQLHSTMT *

DB2 CLI uses each statement handle to relate all the descriptors, result values, cursor information, and status information to the SQL statement processed. Although each SQL statement must have a statement handle, you can reuse the handles for different statements.

The return codes are SQL_SUCCESS, SQL_ERROR, and SQL_INVALID_HANDLE.

SQLPrepare(): Preparing a statement

SQLPrepare() associates an SQL statement with the input statement handle and sends the statement to the DBMS to be prepared. The application can reference this prepared statement by passing the statement handle to other functions.

If the statement handle is used with a **SELECT** statement, **SQLFreeStmt()** must be called to close the cursor before calling **SQLPrepare()**.

Here is the C syntax:

```
SQLRETURN SQLPrepare (SQLHSTMT      hstmt,
                      SQLCHAR       *szSqlStr,
                      SQLINTEGER    cbSqlStr);
```

Table 6-7 on page 212 shows the parameters for the **SQLPrepare()** function.

Table 6-7 Parameters for the SQLPrepare() function

Argument	Description	Use	RPG data type	C data type
hstmt	Statement handle	Input	Integer(10)	SQLHSTMT
*szSqlStr	SQL statement string	Input	Pointer	SQLCHAR *
cbSqlStr	Length of statement string	Input	Integer(5)	SQLINTEGER

A prepared statement may be run once or multiple times by calling **SQLExecute()**. The SQL statement remains associated with the statement handle until the handle is used with another **SQLPrepare()**, **SQLExecDirect()**, **SQLColumns()**, **SQLSpecialColumns()**, **SQLStatistics()**, or **SQLTables()**.

The SQL statement string may contain parameter markers. A parameter marker is represented by a "?" (question mark) character and indicates a position in the statement where the value of an application variable is substituted when **SQLExecute()** is called.

SQLBindParam() is used to bind (or associate) an application variable to each parameter marker and to indicate if any data conversion should be performed at the time the data is transferred.

The return codes are SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, and SQL_INVALID_HANDLE.

SQLBindCol(): Binding a column to an application variable

SQLBindCol() associates (binds) columns in a result set to application variables (storage buffers) for all data types. Data is transferred from the DBMS to the application when **SQLFetch()** is called.

This function is also used to specify any data conversion that is required. It is called once for each column in the result set that the application must retrieve.

SQLPrepare() or **SQLExecDirect()** is usually called before this function. It also might be necessary to call **SQLDescribeCol()** or **SQLColAttributes()**.

SQLBindCol() must be called before **SQLFetch()** to transfer data to the storage buffers that are specified by this call.

Here is the C syntax:

```
SQLRETURN SQLBindCol (SQLHSTMT      hstmt,
                      SQLSMALLINT   icol,
                      SQLSMALLINT   fCType,
                      SQLPOINTER    rgbValue,
                      SQLINTEGER    cbValueMax,
                      SQLINTEGER    *pcbValue);
```

Table 6-8 shows the parameters for the **SQLBindCol()** function.

Table 6-8 Parameters for the SQLBindCol() function

Argument	Description	Use	RPG data type	C data type
hstmt	Statement handle	Input	Integer(10)	SQLHSTMT
icol	Column number	Input	Integer(5)	SQLSMALLINT
fCType	Application data type	Input	Integer(5)	SQLSMALLINT
rgbValue	Pointer to the variable where column data is stored.	Output (defer)	Pointer	SQLPOINTER
cbValueMax	Available size of the variable	Input	Integer(10)	SQLINTEGER
*pcbValue	Size of the data returned in the variable.	Output (defer)	Integer(10)	SQLINTEGER *

For this function, both **rgbValue** and **pcbValue** are deferred outputs, meaning that the storage locations to which these pointers point are not updated until **SQLFetch()** is called. The locations that are referred to by these pointers must remain valid until **SQLFetch()** is called.

The application calls **SQLBindCol()** once for each column in the result set that it wants to retrieve. When **SQLFetch()** is called, the data in each of these bound columns is placed in the assigned location (given by the pointers **rgbValue** and **pcbValue**). Columns are identified by a number, assigned sequentially from left to right, starting at 1.

The application must ensure enough storage is allocated for the data to be retrieved. If the data type is either SQL_CHAR or SQL_DEFAULT, the available size of the variable (cbValueMax) must be greater than 0. If the data type is either SQL_DECIMAL or SQL_NUMERIC, the size of the variable can be determined by using the following formula:

$$(\text{Precision} * 256) + \text{scale}$$

For example, the size of variable for a numeric field, declared as (6,2), calculated by using this formula, is:

$$6 * 256 + 2 = 1538$$

The application can query the attributes (such as data type and length) of the column by first calling **SQLDescribeCol()** or **SQLColAttributes()**. This information can then be used to specify the correct data type of the storage locations or to indicate data conversion to other data types.

Return codes are SQL_SUCCESS, SQL_ERROR, and SQL_INVALID_HANDLE.

SQLBindParameter(): Binding a buffer to a parameter marker

SQLBindParameter() binds an application variable to a parameter marker in an SQL statement. This function can also be used to bind an application variable to a parameter of a stored procedure **CALL** statement where the parameter may be input or output. This function is the same as **SQLSetParam()**.

Here is the C syntax:

```
SQLRETURN SQLBindParameter (SQLHSTMT      hstmt,
                           SQLSMALLINT    ipar,
                           SQLSMALLINT    InputOutputType,
                           SQLSMALLINT    valueType,
                           SQLSMALLINT    ParameterType,
                           SQLINTEGER     columnSize,
                           SQLSMALLINT    ibScale,
                           SQLPOINTER     rgbValue,
                           SQLINTEGER     bufferLength
                           SQLINTEGER    *pcbValue);
```

Table 6-9 shows the parameters for the **SQLBindParameter()** function.

Table 6-9 Parameters for the SQLBindParameter() function

Argument	Description	Use	RPG data type	C data type
hstmt	Statement handle	Input	Integer(10)	SQLHSTMT
ipar	Parameter marker number	Input	Integer(5)	SQLSMALLINT
InputOutputType	Application data type	Input	Integer(5)	SQLSMALLINT
ValueType	C data type	Input	Integer(5)	SQLSMALLINT
ParameterType	SQL data type	Input	Integer(5)	SQLSMALLINT
columnSize	Precision of parameter	Input	Integer(10)	SQLINTEGER
ibScale	Number of decimal positions	Input	Integer(5)	SQLSMALLINT
rgbValue	Buffer with actual data for the parameter	Input or Output	Pointer	SQLPOINTER
BufferLength	Length of buffer	Input	Integer(10)	SQLINTEGER

Argument	Description	Use	RPG data type	C data type
*pbcValue	Value that is interpreted during execution	Input	Integer(10)	SQLINTEGER *

The return codes are SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, and SQL_INVALID_HANDLE.

SQLExecute(): Running a statement

SQLExecute() runs a statement that was successfully prepared by using **SQLPrepare()** once or multiple times. The statement is run by using the current values of any application variables that were bound to parameter markers by **SQLBindParam()**.

Here is the C syntax:

```
SQLRETURN SQLExecute (SQLHSTMT      hstmt);
```

Table 6-10 shows the parameters for the **SQLExecute()** function.

Table 6-10 Parameters for the SQLExecute() function

Argument	Description	Use	RPG data type	C data type
hstmt	Statement handle	Input	Integer(10)	SQLHSTMT

The SQL statement string may contain parameter markers. A parameter marker is represented by a "?" (question mark) character and indicates a position in the statement where the value of an application variable is substituted when **SQLExecute()** is called.

SQLBindParam() is used to bind (or associate) an application variable to each parameter marker and to indicate whether any data conversion should be performed at the time the data is transferred.

All parameters must be bound before calling **SQLExecute()**. After the application processes the results from the **SQLExecute()** call, it can run the statement again with new (or the same) values in the application variables.

The return codes are SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, SQL_INVALID_HANDLE, and SQL_NO_DATA_FOUND.

SQLExecDirect(): Running a statement directly

SQLExecDirect() directly runs the specified SQL statement. The statement can be run only once. Also, the connected database server must be able to prepare the statement.

Here is the C syntax:

```
SQLRETURN SQLExecDirect (SQLHSTMT      hstmt,
                        SQLCHAR        *szSqlStr,
                        SQLINTEGER     cbSqlStr);
```

Table 6-11 shows the parameters for the **SQLExecDirect()** function.

Table 6-11 Parameters for the SQLExecDirect() function

Argument	Description	Use	RPG data type	C data type
hstmt	Statement handle	Input	Integer(10)	SQLHSTMT
*szSqlStr	SQL statement string	Input	Pointer	SQLCHAR *

Argument	Description	Use	RPG data type	C data type
cbSqlStr	Length of the statement string	Input	Integer(5)	SQLINTEGER

The SQL statement string may contain parameter markers. A parameter marker is represented by a "?" (question mark) character, and indicates a position in the statement where the value of an application variable is substituted when **SQLExecDirect()** is called. **SQLBindParam()** binds (or associates) an application variable to each parameter marker to indicate whether any data conversion should be performed at the time the data is transferred. All parameters must be bound before calling **SQLExecDirect()**.

The return codes are SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, SQL_INVALID_HANDLE, and SQL_NO_DATA_FOUND.

SQLFetch(): Fetching the next row

SQLFetch() advances the cursor to the next row of the result set and retrieves any bound columns.

SQLFetch() can be used to receive the data directly into variables that you specify with **SQLBindCol()**, or the columns can be received individually after the fetch by calling **SQLGetData()**. Data conversion is also performed when **SQLFetch()** is called, if conversion was indicated when the column was bound.

Here is the C syntax:

```
SQLRETURN SQLFetch (SQLHSTMT hstmt);
```

Table 6-12 shows the parameters for the **SQLFetch()** function.

Table 6-12 Parameters for the SQLFetch() function

Argument	Description	Use	RPG data type	C data type
hstmt	Statement handle	Input	Integer(10)	SQLHSTMT

SQLFetch() can be called only if the most recently run statement on **hstmt** was a **SELECT**.

The number of application variables bound with **SQLBindCol()** must not exceed the number of columns in the result set, or **SQLFetch()** fails.

If **SQLBindCol()** is not called to bind any columns, then **SQLFetch()** does not return data to the application, but advances the cursor. In this case, **SQLGetData()** can be called to obtain all of the columns individually. Data in unbound columns is discarded when **SQLFetch()** advances the cursor to the next row.

When all the rows are retrieved from the result set, or the remaining rows are not needed, **SQLFreeStmt()** should be called to close the cursor and discard the remaining data and associated resources.

The return codes are SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, SQL_INVALID_HANDLE, and SQL_NO_DATA_FOUND.

SQLTransact(): Transaction management

SQLTransact() commits or rolls back the current transaction in the connection. All changes to the database that are performed on the connection since the connect time or the previous call to **SQLTransact()** (whichever is most recent) are committed or rolled back.

If a transaction is active on a connection, the application must call **SQLTransact()** before it can disconnect from the database.

Here is the C syntax:

```
SQLRETURN SQLTransact (SQLHENV      henv,
                      SQLHDBC      hdbc,
                      SQLSMALLINT   fType);
```

Table 6-13 shows the parameters for the **SQLTransact()** function.

Table 6-13 Parameters for the SQLTransact() function

Argument	Description	Use	RPG data type	C data type
henv	Environment handle	Input	Integer(10)	SQLHENV
hdbc	Database connection handle	Input	Integer(10)	SQLHDBC
fType	Desired action for transaction	Input	Integer(5)	SQLSMALLINT

Completing a transaction with **SQL_COMMIT** or **SQL_ROLLBACK** has the following effects:

- ▶ Prepared SQL statements do not survive transactions. The application must prepare statements again to run them as part of a new transaction, which means that statement handles are still valid after a call to **SQLTransact()**, and can be reused for later SQL statements or deallocated by calling **SQLFreeStmt()**.
- ▶ Cursor names, bound parameters, and column bindings survive transactions.
- ▶ Open cursors are closed, and any result sets that are pending retrieval are discarded.

The return codes are **SQL_SUCCESS**, **SQL_ERROR**, and **SQL_INVALID_HANDLE**.

SQLError(): Retrieving error information

SQLError() returns the diagnostic information that is associated with the most recently called DB2 CLI function for a particular statement, connection, or environment handle. The information consists of a standardized SQLSTATE, native error code, and a text message.

Call **SQLError()** after receiving a return code of **SQL_ERROR** or **SQL_SUCCESS_WITH_INFO** from another function call.

Here is the C syntax:

```
SQLRETURN SQLError (SQLHENV      henv,
                    SQLHDBC      hdbc,
                    SQLHSTMT     hstmt,
                    SQLCHAR       *szSqlState,
                    SQLINTEGER    *pfNativeError,
                    SQLCHAR       *szErrorMsg,
                    SQLSMALLINT   cbErrorMsgMax,
                    SQLSMALLINT   *pcbErrorMsg);
```

Table 6-14 shows the parameters for the **SQLError()** function.

Table 6-14 Parameters for the SQLError() function

Argument	Description	Use	RPG data type	C data type
henv	Environment handle	Input	Integer(10)	SQLHENV
hdbc	Database connection handle	Input	Integer(10)	SQLHDBC

Argument	Description	Use	RPG data type	C data type
hstmt	Statement handle	Input	Integer(10)	SQLHSTMT
*szSqlState	SQLSTATE as a string	Output	Pointer	SQLCHAR *
*pfNativeError	Native error code (SQLCODE)	Output	Integer(10)	SQLINTEGER *
*szErrorMsg	Pointer to buffer with message text	Output	Pointer	SQLCHAR *
cbErrorMsgMax	Maximum length of error message	Input	Integer(5)	SQLSMALLINT
*pcbErrorMsg	Total length of error message	Output	Integer(5)	SQLSMALLINT

Use the following methods to obtain diagnostic information:

- ▶ For an environment, pass a valid environment handle. Set **hdbc** and **hstmt** to SQL_NULL_HDBC and SQL_NULL_HSTMT, respectively.
- ▶ For a connection, pass a valid database connection handle, and set **hstmt** to SQL_NULL_HSTMT. The **henv** argument is ignored.
- ▶ For a statement, pass a valid statement handle. The **henv** and **hdbc** arguments are ignored.

If diagnostic information that is generated by one DB2 CLI function is not retrieved before a function other than **SQLError()** is called with the same handle, the information for the previous function call is lost. This is true whether diagnostic information is generated for the second DB2 CLI function call.

To avoid truncation of the error message, declare a buffer length of SQL_MAX_MESSAGE_LENGTH + 1. The message text is never longer than this.

The return codes are SQL_SUCCESS, SQL_ERROR, SQL_INVALID_HANDLE, and SQL_NO_DATA_FOUND.

SQLFreeStmt(): Freeing (or resetting) a statement handle

SQLFreeStmt() ends processing on the statement that is referenced by the statement handle. Use this function to complete the following tasks:

- ▶ Close a cursor.
- ▶ Reset parameters.
- ▶ Unbind columns from variables.
- ▶ Drop the statement handle and free the DB2 CLI resources that are associated with the statement handle.

SQLFreeStmt() is called after running an SQL statement and processing the results.

Here is the C syntax:

```
SQLRETURN SQLFreeStmt (SQLHSTMT      hstmt,
                      SQLSMALLINT   fOption);
```

Table 6-15 shows the parameters for the **SQLFreeStmt()** function.

Table 6-15 Parameters for the SQLFreeStmt() function

Argument	Description	Use	RPG data type	C data type
hstmt	Statement handle	Input	Integer(10)	SQLHSTMT
fOption	Mode of deallocation	Input	Integer(5)	SQLSMALLINT

SQLFreeStmt() can be called with the following options:

- ▶ **SQL_CLOSE**

The cursor (if any) that is associated with the statement handle (**hstmt**) is closed and all pending results are discarded. The application can reopen the cursor by calling **SQLExecute()** with the same or different values in the application variables (if any) that are bound to **hstmt**.

- ▶ **SQL_DROP**

DB2 CLI resources that are associated with the input statement handle are freed, and the handle is invalidated. The open cursor, if any, is closed and all pending results are discarded.

- ▶ **SQL_UNBIND**

All the columns that are bound by previous **SQLBindCol()** calls on this statement handle are released.

- ▶ **SQL_RESET_PARAMS**

All the parameters that are set by previous **SQLBindParam()** calls on this statement handle are released. The association between application variables or file references and parameter markers in the SQL statement of the statement handle is broken.

The return codes are **SQL_SUCCESS**, **SQL_SUCCESS_WITH_INFO**, **SQL_ERROR**, and **SQL_INVALID_HANDLE**.

SQLDisconnect(): Disconnecting from a data source

SQLDisconnect() closes the connection that is associated with the database connection handle. After calling this function, either call **SQLConnect()** to connect to another database or call **SQLFreeConnect()**.

Here is the C syntax:

```
SQLRETURN SQLDisconnect (SQLHDBC      hdbc);
```

Table 6-16 shows the parameters for the **SQLDisconnect()** function.

Table 6-16 Parameters for the SQLDisconnect() function

Argument	Description	Use	RPG data type	C data type
hdbc	Connection handle	Input	Integer(10)	SQLHDBC

If an application calls **SQLDisconnect()** before it has freed all the statement handles that are associated with the connection, DB2 CLI frees them after it successfully disconnects from the database.

After a successful **SQLDisconnect()** call, the application can re-use **hdbc** to make another **SQLConnect()** request.

The return codes are SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, and SQL_INVALID_HANDLE.

SQLFreeConnect(): Free connection handle

SQLFreeConnect() invalidates and frees the connection handle. All DB2 CLI resources that are associated with the connection handle are freed. **SQLDisconnect()** must be called before calling this function. Either **SQLFreeEnv()** is called next to continue terminating the application or **SQLAllocHandle()** to allocate a new connection handle.

Here is the C syntax:

```
SQLRETURN SQLFreeConnect (SQLHDBC      hdbc);
```

Table 6-17 shows the parameters for the **SQLFreeConnect()** function.

Table 6-17 Parameters for the SQLFreeConnect() function

Argument	Description	Use	RPG data type	C data type
hdbc	Connection handle	Input	Integer(10)	SQLHDBC

If this function is called when a connection still exists, SQL_ERROR is returned, and the connection handle remains valid.

Return codes are SQL_SUCCESS, SQL_ERROR, and SQL_INVALID_HANDLE.

SQLFreeEnv(): Freeing the environment handle

SQLFreeEnv() invalidates and frees the environment handle. All DB2 CLI resources that are associated with the environment handle are freed.

SQLFreeConnect() must be called before calling this function.

This function is the last DB2 CLI step an application needs before terminating.

Here is the C syntax:

```
SQLRETURN SQLFreeEnv (SQLHENV      henv);
```

Table 6-18 shows parameters for the **SQLFreeEnv()** function.

Table 6-18 Parameters for the SQLFreeEnv() function

Argument	Description	Use	RPG data type	C data type
henv	Environment handle	Input	Integer(10)	SQLHENV

If this function is called when there is still a valid connection handle, SQL_ERROR is returned, and the environment handle remains valid.

Return codes are SQL_SUCCESS, SQL_ERROR, and SQL_INVALID_HANDLE.

6.4.8 Introduction to a CLI example

To illustrate the use of CLI interface, this section uses a modified version of the example that is shown in 6.2.6, “Source code for the SQLEMBED program” on page 184. Embedded SQL statements are replaced with subprocedures that call CLI functions.

All subprocedure prototypes that are required for this example are made available through the include file CLIPRTO. This file contains only selected functions that are used in the example. It also contains definitions of constants for CLI return codes, connect attributes, SQL data types, and other options used by CLI functions.

Source code for the SQLCLI program

The logic of the program SQLCLI is the same as that of the programs that are used in 6.2.6, “Source code for the SQLEMBED program” on page 184, and 6.3.4, “A stored procedure example” on page 191. This section demonstrates that the same functions can be achieved by using either embedded SQL, stored procedures, or DB2 CLI. The program uses the display file DSPFIL1 to communicate with the user.

To create this program, two steps are required:

1. Create the module by running the CL command **CRTRPGMOD**.
2. Create the program by running the CL command **CRTPGM** with the parameter **BNDSRVPGM(QSYS/QSQCLI)**.

Example 6-24 show the source code for the SQLCLI program.

Example 6-24 Source code for the SQLCLI program

```

//*****
// source member SQLCLI from DBSRC in RPGISCOOL
// Simple ILE RPG program SQLCLI to test CLI interface
//
// Examples of SELECT, INSERT, UPDATE, DELETE using CLI
//
// 1. Compile this source member as module SQLCLI (PDM Option=15)
//
// 2. Create program SQLCLI from module SQLCLI (PDM Option=26)
//    with PROMPT(PF4) and BNDSRVPGM(QSYS/QSQCLI)
//*****-
//-----+
// The information contained in this document has not been submitted
// to any formal IBM test and is distributed AS IS. 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. See Special Notices in redbook SG24-5402 for more.
//-----+
// Display file with subfile
dcl-F DspFill WorkStn(*ext) IndDS(DispInd) SFile(Sf1Rec1:RecNum);

//-----+
Dcl-s RecNum      zoned(3 : 0);

// Indicator data structure for display file indicators
//
Dcl-ds DispInd;

```

```

        Exit           ind pos(3);
        ListAllRec    ind pos(4);
        InsertRec     ind pos(5);
        UpdateRec     ind pos(6);
        DeleteRec     ind pos(7);
        Cancel         ind pos(12);
        InvalidRec    ind pos(55);
        ClearSfl      ind pos(66);
end-Ds;

// Include copy member with CLI prototypes and SQL variables
//1
/COPY RPGISCOOL/DBSRC,CLIPROTO

// Program variables used by CLI interface
//2
Dcl-s SQL_RC    int(10);
Dcl-s henv       int(10);
Dcl-s hdbc       int(10);
Dcl-s DBName     char(18);
Dcl-s DBNameP    pointer INZ(%ADDR(DBName));
Dcl-s DBUser     char(10);
Dcl-s DBUserP   pointer INZ(%ADDR(DBUser));
Dcl-s DBPwd      char(10);
Dcl-s DBPwdP    pointer INZ(%ADDR(DBPwd));
Dcl-s ConnOpt    int(5);
Dcl-s Is1Lvl     int(10);
Dcl-s Is1LvlP   pointer INZ(%ADDR(Is1Lvl));
Dcl-s hstmt      int(10);
Dcl-s SQLStm    char(128);
Dcl-s SQLStmP   pointer INZ(%ADDR(SQLStm));
Dcl-s SQLState   char(6);
Dcl-s SQLStateP  pointer INZ(%ADDR(SQLState));
Dcl-s SQLCode    int(10);
Dcl-s SQLMsg     char(71);
Dcl-s SQLMsgP   pointer INZ(%ADDR(SQLMsg));
Dcl-s MsgLenMax int(5);
Dcl-s MsgLen     int(5);
Dcl-s ZeroBin    char(128) INZ(*ALLX'00');
Dcl-s ParamLen   int(10);
Dcl-s ParamDec   int(5);
Dcl-s pcbValue   int(10) INZ(0);
Dcl-s Param1P    pointer;
Dcl-s Param2P    pointer;
Dcl-s Param3P    pointer;
Dcl-s Param4P    pointer;
Dcl-s Param5P    pointer;

//-----
// Begin of CLI initialization tasks
//3
MsgLenMax=SQL_MAXMSG+1;
// Allocate environment handle
SQL_RC=SQLAllocEnv(henv);4
// Allocate connection handle

```

```

SQL_RC=SQLAllocCon(henv:hdhc);                                5
// Set connection options
ConnOpt=SQL_ISOLVL;
IsLvl=SQL_NONE;
SQL_RC=SQLSetCnOp(hdbc:ConnOpt:IsLvlP);

// Connect to database
DBName='dbname';                                              6
DBName=(%TRIM(DBName))+Zerobin;
DBUser='username';
DBUser=(%TRIM(DBUser))+Zerobin;
DBPwd='password';
DBPwd=(%TRIM(DBPwd))+Zerobin;
SQL_RC=SQLConnect(hdbc:DBNameP:SQL_NTS:
                  DBUserP:SQL_NTS:DBPwdP:SQL_NTS);

// Check CLI Initialization
If SQL_RC<>SQL_OK;                                         7
  SQL_RC=SQLGetError(henv:hdhc:hstmt:SQLStateP:
                      SQLCode:SQLMsgP:MsgLenMax:MsgLen);
EndIf;

// End of CLI initialization tasks
//-----
Exfmt DspRec1;
// Loop begin
DoW Not (Exit Or Cancel);
  InvalidRec=*Off;
  Select;
    // * * Insert record
  When InsertRec;
    Clear DspDes;
    Clear DspQty;
    Clear DspPrc;
    Clear DspDat;
    DspNum=PartNo;
    Exfmt DspRec2;
    // Allocate statement handle for INSERT
    SQL_RC=SQLAllocStmt(hdbc:hstmt);                           9
    // Bind value for 1. parameter marker PartNum
    Param1P=%ADDR(DspNum);                                     10
    ParamLen=5;
    ParamDec=0;
    SQL_RC=SQLBindPar(hstmt:1:SQL_DECIM:
                      SQL_DECIM:ParamLen:ParamDec:Param1P:
                      pcbValue);

    // Bind value for 2. parameter marker PartDes
    Param2P=%ADDR(DspDes);                                     10
    ParamLen=25;
    ParamDec=0;
    SQL_RC=SQLBindPar(hstmt:2:SQL_CHAR:
                      SQL_CHAR:ParamLen:ParamDec:Param2P:
                      pcbValue);

    // Bind value for 3. parameter marker PartQty

```

```

Param3P=%ADDR(DspQty);                                10
ParamLen=5;
ParamDec=0;
SQL_RC=SQLBindPar(hstmt:3:SQL_DECIM:
    SQL_DECIM:ParamLen:ParamDec:Param3P:
    pcbValue);
// Bind value for 4. parameter marker PartPrc
Param4P=%ADDR(DspPrc);                               10
ParamLen=6;
ParamDec=2;
SQL_RC=SQLBindPar(hstmt:4:SQL_DECIM:
    SQL_DECIM:ParamLen:ParamDec:Param4P:
    pcbValue);
// Bind value for 5. parameter marker PartDat
Param5P=%ADDR(DspDat);                               10
ParamLen=10;
ParamDec=0;
SQL_RC=SQLBindPar(hstmt:5:SQL_CHAR:
    SQL_DATE:ParamLen:ParamDec:Param5P:
    pcbValue);
// Define and execute INSERT statement
SQLStm='INSERT INTO RPGISCOOL.PARTS '
    + 'VALUES(?, ?, ?, ?, ?)';
SQLStm=(%TRIM(SQLStm))+Zerobin;
SQL_RC=SQLExecDir(hstmt:SQLStmP:SQL_NTS);
// Check for duplicate record error
If SQL_RC<>SQL_OK;
    InvalidRec=*On;
Endif;
// Free statement handle for INSERT
SQL_RC=SQLFreeStm(hstmt:SQL_DROP);                  14
// * * Display all records
When ListAllRec;
// Clear subfile
RecNum=0;
ClearSfl=*On;
Write SflRec2;
ClearSfl=*Off;
// Allocate statement handle for SELECT all records
SQL_RC=SQLAllocStmt(hdbc:hstmt);                     9
// Define and execute SELECT statement
SQLStm='SELECT * FROM RPGISCOOL.PARTS '
    + 'ORDER BY PARTNUM';
SQLStm=(%TRIM(SQLStm))+Zerobin;
SQL_RC=SQLExecDir(hstmt:SQLStmP:SQL_NTS);
// Bind 1. column to program variable DspNum
Param1P=%ADDR(DspNum);                            11
ParamLen=1280;
SQL_RC=SQLBindCol(hstmt:1:SQL_DECIM:
    Param1P:ParamLen:pcbValue);
// Bind 2. column to program variable DspDes
Param2P=%ADDR(DspDes);                           12
ParamLen=25;
SQL_RC=SQLBindCol(hstmt:2:SQL_CHAR:
    Param2P:ParamLen:pcbValue);

```

```

// Bind 3. column to program variable DspQty
Param3P=%ADDR(DspQty);
ParamLen=1280;
SQL_RC=SQLBindCol(hstmt:3:SQL_DECIM:
    Param3P:ParamLen:pcbValue);
// Bind 4. column to program variable DspPrc
Param4P=%ADDR(DspPrc);
ParamLen=1538;
SQL_RC=SQLBindCol(hstmt:4:SQL_DECIM:
    Param4P:ParamLen:pcbValue);
// Bind 5. column to program variable DspDat
Param5P=%ADDR(DspDat);
ParamLen=10;
SQL_RC=SQLBindCol(hstmt:5:SQL_DATE:
    Param5P:ParamLen:pcbValue);
// Execute Fetch in loop to read all records
DoU SQL_RC=SQL_NODATA;
SQL_RC=SQLFetch(hstmt);
If SQL_RC=SQL_NODATA;
Leave;
EndIf;
RecNum=RecNum+1;
Write SflRec1;
EndDo;
// Free statement handle for SELECT all records
SQL_RC=SQLFreeStm(hstmt:SQL_DROP);  
14
Exfmt SflRec2;
// * * Display single record
Other;
// Allocate statement handle for SELECT single record
SQL_RC=SQLAllocStmt(hdbc:hstmt);  
9
// Bind value for 1. parameter marker PartNum
Param1P=%ADDR(PartNo);  
10
ParamLen=5;
ParamDec=0;
SQL_RC=SQLBindPar(hstmt:1:SQL_DECIM:
    SQL_DECIM:ParamLen:ParamDec:Param1P:
    pcbValue);
// Define and execute SELECT statement
SQLStm='SELECT PARTDES, PARTQTY, '+  
11
    'PARTPRC, PARTDAT FROM RPGISCOOL.PARTS '
    + 'WHERE PARTNUM=?';
SQLStm=(%TRIM(SQLStm))+Zerobin;
SQL_RC=SQLExecDir(hstmt:SQLStmP:SQL_NTS);
// Bind 1. column to program variable DspDes
Param2P=%ADDR(DspDes);  
12
ParamLen=25;
SQL_RC=SQLBindCol(hstmt:1:SQL_CHAR:
    Param2P:ParamLen:pcbValue);
// Bind 2. column to program variable DspQty
Param3P=%ADDR(DspQty);
ParamLen=1280;
SQL_RC=SQLBindCol(hstmt:2:SQL_DECIM:
    Param3P:ParamLen:pcbValue);
// Bind 3. column to program variable DspPrc

```

```

Param4P=%ADDR(DspPrc);
ParamLen=1538;
SQL_RC=SQLBindCol(hstmt:3:SQL_DECIM:
    Param4P:ParamLen:pcbValue);
// Bind 4. column to program variable DspDat
Param5P=%ADDR(DspDat);
ParamLen=10;
SQL_RC=SQLBindCol(hstmt:4:SQL_DATE:
    Param5P:ParamLen:pcbValue);
// Execute Fetch to read single record
SQL_RC=SQLFetch(hstmt);                                         12
If SQL_RC=SQL_NODATA;
    InvalidRec=*On;
EndIf;
// Free statement handle for SELECT single record
SQL_RC=SQLFreeStm(hstmt:SQL_DROP);                                13
// Check for no record found error
If Not InvalidRec;
    DspNum=PartNo;
    Exfmt DspRec2;
    Select;
    When Exit Or Cancel;
        Leave;
        // * * Update record
    When UpdateRec;
        // Allocate statement handle for UPDATE
        SQL_RC=SQLAllocStmt(hdbc:hstmt);                               9
        // Bind value for 1. parameter marker PartDes
        Param1P=%ADDR(DspDes);                                         10
        ParamLen=25;
        ParamDec=0;
        SQL_RC=SQLBindPar(hstmt:1:SQL_CHAR:
            SQL_CHAR:ParamLen:ParamDec:Param1P:
            pcbValue);
        // Bind value for 2. parameter marker PartQty
        Param2P=%ADDR(DspQty);                                         10
        ParamLen=5;
        ParamDec=0;
        SQL_RC=SQLBindPar(hstmt:2:SQL_DECIM:
            SQL_DECIM:ParamLen:ParamDec:Param2P:
            pcbValue);
        // Bind value for 3. parameter marker PartPrc
        Param3P=%ADDR(DspPrc);                                         10
        ParamLen=6;
        ParamDec=2;
        SQL_RC=SQLBindPar(hstmt:3:SQL_DECIM:
            SQL_DECIM:ParamLen:ParamDec:Param3P:
            pcbValue);
        // Bind value for 4. parameter marker PartDat
        Param4P=%ADDR(DspDat);                                         10
        ParamLen=10;
        ParamDec=0;
        SQL_RC=SQLBindPar(hstmt:4:SQL_CHAR:
            SQL_DATE:ParamLen:ParamDec:Param4P:
            pcbValue);

```

```

// Bind value for 5. parameter marker PartNum
Param5P=%ADDR(DspNum);
ParamLen=5;
ParamDec=0;
SQL_RC=SQLBindPar(hstmt:5:SQL_DECIM:
    SQL_DECIM:ParamLen:ParamDec:Param5P:
    pcbValue);
// Define and execute UPDATE statement
SQLStm='UPDATE RPGISCOOL.PARTS SET '
    + 'PARTDES=? , PARTQTY=? , PARTPRC=? , '
    + 'PARTDAT=? WHERE PARTNUM=?';
SQLStm=(%TRIM(SQLStm))+Zerobin;
SQL_RC=SQLEExecDir(hstmt:SQLStmP:SQL_NTS);
// Free statement handle for UPDATE
SQL_RC=SQLFreeStm(hstmt:SQL_DROP);          14
// * * Delete record
When DeleteRec;
    // Allocate statement handle for DELETE
    SQL_RC=SQLA1cStmt(hdbc:hstmt);           9
    // Bind value for 1. parameter marker PartNum
    Param1P=%ADDR(PartNo);                  10
    ParamLen=5;
    ParamDec=0;
    SQL_RC=SQLBindPar(hstmt:1:SQL_DECIM:
        SQL_DECIM:ParamLen:ParamDec:Param1P:
        pcbValue);
    // Define and execute DELETE statement
    SQLStm='DELETE FROM RPGISCOOL.PARTS '
        + 'WHERE PARTNUM=?';
    SQLStm=(%TRIM(SQLStm))+Zerobin;
    SQL_RC=SQLEExecDir(hstmt:SQLStmP:SQL_NTS);
    // Free statement handle for DELETE
    SQL_RC=SQLFreeStm(hstmt:SQL_DROP);          14
    EndS1;
    EndIf;
EndS1;
Exfmt DspRec1;
// Loop end
EndDo;

-----
// Begin of CLI Termination tasks
// Disconnect form database
SQL_RC=SQLDisconnect(hdbc);                  15
// Free connection handle
SQL_RC=SQLFreeCon(hdbc);                     16
// Free environment handle
SQL_RC=SQLFreeEnv(henv);                     17
// End of CLI Termination tasks
//-----
*INLR = *On;

```

SQLCLI program notes

The notes in this section refer to the numbers shown on the right side of Example 6-24 on page 221:

1. Copy the source member CLIPROTO to include all required subprocedure prototypes.
2. Variables that are used by different CLI functions.
3. To avoid truncation of the error message, declare a message buffer length as SQL_MAXMSG + 1. The message text is never longer than this setting.
4. The function **SQLAllocEnv()** allocates an environment handle and associated resources. This function must be called before any other CLI function.
5. The function **SQLAllocConnect()** allocates a connection handle and associated resources within the environment that is identified by the input environment handle.
6. The function **SQLSetConnectOption()** sets connection attributes for a particular connection. With SQL_NONE, you define that commitment control is not used.
7. The function **SQLConnect()** establishes a connection to the target database, which must already be defined on the system for the connection to work. You can use the Work with Relational Database Directory Entries (**WRKRDBDIRE**) command to determine which databases already are defined and optionally define an additional one.
8. An example of how to check the success of an executed CLI function. **SQLError()** returns the diagnostic information that is associated with the most recently called CLI function for a particular statement, connection, or environment handle.
9. The function **SQLAllocStmt()** allocates a new statement handle and associates it with the connection that is specified by the connection handle. In this example, there is only one statement handle, which is allocated to different SQL statements. Another possibility is to allocate a separate handle for each SQL statement.
10. The function **SQLBindParam()** binds a parameter marker to a program variable. A parameter marker is represented by a "?" (question mark) character in an SQL statement and is used to indicate a position in the statement where an application supplied value is substituted when the statement is run. Parameter markers are referred to by number and are numbered sequentially from left to right, starting at 1.
11. An SQL statement string, including parameter markers, is defined in the variable SQLStm, which is then passed to the function **SQLExecDirect()** to be run. All parameters must be bound before calling.
12. The function **SQLBindCol()** binds columns in a result set to program variables for all data types. Data is transferred from the DBMS to the application when **SQLFetch()** is called.
13. When the executed SQL statement is **SELECT**, it is necessary to run the function **SQLFetch()**, which advances the cursor to the next row of the result set and retrieves any bound columns.
14. The function **SQLFreeStmt()** ends processing on the statement that is referenced by the statement handle. It is used to close a cursor, reset parameters, and unbind columns from variables.
15. The function **SQLDisconnect()** closes the connection that is associated with the database connection handle. This is part of the termination process.
16. The function **SQLFreeConnect()** is called next to free the connection handle. All CLI resources that are associated with the connection handle are freed.
17. The function **SQLFreeEnv()** is the last CLI step an application needs before terminating. It invalidates and frees the environment handle.

Source code for CLIPROTO prototypes

This source member is copied into the previous SQLCLI program. It contains subprocedure prototypes for CLI functions used by this program.

Example 6-25 shows the source code for the CLIPROTO prototypes.

Example 6-25 Source code for CLIPROTO prototypes

```

/IF DEFINED(CLIPROTO_Included)
  /EOF
/ENDIF
/DEFINE CLIPROTO_Included
//*****
// Source member CLIPROTO from DBSRC in RPGISCOOL
// Function Prototype ALLOCATE ENVIRONMENT HANDLE
//
//      SQLRETURN SQLAllocEnv (SQLHENV *phenv);
//*****
//-----
// The information contained in this document has not been submitted
// to any formal IBM test and is distributed AS IS. 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. See Special Notices in redbook SG24-5402 for more.
//-----
// Return value = 0 (OK) or -1 (error)
Dcl-pr SQLAllocEnv int(10) ExtProc('SQLAllocEnv');
// Environmental handle
  envHndl int(10);
end-Pr;

//*****
// Function Prototype ALLOCATE CONNECTION HANDLE
//
//      SQLRETURN SQLAllocConnect (SQLHENV henv,
//                                SQLHDBC *phdbc);
//*****
// Return value = 0 (OK) or -1 (error) or -2 (invalid handle)
Dcl-pr SQLAllocCon int(10) ExtProc('SQLAllocConnect');
// Environmental handle
  envHndl int(10) value;
// Connection handle
  connHndl int(10);
end-Pr;

//*****
// Function Prototype CONNECTION TO A DATABASE
//
//      SQLRETURN SQLConnect (SQLHDBC      hdbc,
//                            SQLCHAR       *szDSN,
//                            SQLSMALLINT   cbDSN,
//                            SQLCHAR       *szUID,
//                            SQLSMALLINT   cbUID,
//                            SQLCHAR       *szAuthStr,
//                            SQLSMALLINT   cbAuthStr);

```

```

//*****
// Return value = 0 or 1 (OK) or -1 (error) or -2 (invalid handle)
Dcl-pr SQLConnect  int(10) ExtProc('SQLConnect');
// Connection handle
connHndle      int(10) value;
// Pointer to the field containing the name of the database
DBName          pointer value;
// Length of the name of the database
DBNameLength    int(5) value;
// Pointer to the field containing the user identification
userIDName     pointer value;
// Length of the user identification
userIDNameLength int(5) value;
// Pointer to the field containing the password
password        pointer value;
// Length of the password
passwordNameLength int(5) value;
end-Pr;

//*****
// Function Prototype SET CONNECTION OPTION
//
//      SQLRETURN SQLSetConnectOption (SQLHDBC      hdbc,
//                                      SQLSMALLINT   fOption,
//                                      SQLPOINTER    vParam);
//*****
// Return value = 0 or 1 (OK) or -1 (error) or -2 (invalid handle)
Dcl-pr SQLSetCnOp  int(10) ExtProc('SQLSetConnectOption');
// Connection handle
connHndle      int(10) value;
// Connect option
connectOption   int(5) value;
// Pointer to the field containing the value of the connect option
option          pointer value;
end-Pr;

//*****
// Function Prototype ALLOCATE STATEMENT HANDLE
//
//      SQLRETURN SQLAllocStmt (SQLHDBC      hdbc,
//                             SQLHSTMT *phstmt);
//*****
// Return value = 0 (OK) or -1 (error) or -2 (invalid handle)
Dcl-pr SQLAllocStmt int(10) ExtProc('SQLAllocStmt');
// Connection handle
connHndle      int(10) value;
// Handle of the SQL statement
SQLHndl       int(10);
end-Pr;

//*****
// Function Prototype PREPARE SQL STATEMENT
//
//      SQLRETURN SQLPrepare (SQLHSTMT      hstmt,
//                            SQLCHAR       *szSqlStr,

```

```

//                                     SQLINTEGER cbSqlStr);
//*****
// Return value = 0 or 1 (OK) or -1 (error) or -2 (invalid handle)
Dcl-pr SQLPrepare  int(10) ExtProc('SQLPrepare');
// Handle of the SQL statement
SQLHndl      int(10) value;
// Pointer to the field containing the SQL statement
SQLStmt      pointer value;
// Length of the SQL statement
SQLStmtLength int(5) value;
end-Pr;

//*****
// Function Prototype BIND BUFFER TO A PARAMETER MARKER
//
//          SQLRETURN SQLBindParameter(SQLHSTMT    hstmt,
//                                 SQLSMALLINT iparm,
//                                 SQLSMALLINT iType,
//                                 SQLSMALLINT vType,
//                                 SQLSMALLINT pType,
//                                 SQLINTEGER pLen,
//                                 SQLSMALLINT pScale,
//                                 SQLPOINTER pData,
//                                 SQLINTEGER BLen,
//                                 SQLINTEGER *pcbValue);
//*****
// Return value = 0 or 1 (OK) or -1 (error) or -2 (invalid handle)
Dcl-pr SQLBindPar  int(10) ExtProc('SQLBindParameter');
// Handle of the SQL statement
SQLHndl      int(10) value;
// Sequential parameter marker number
parmMarker   int(5) value;
// Input/output type of the parameter (application)
appDataType  int(5) value;
// Value type of the parameter (SQL)
SQLValueType int(5) value;
// Data type of the parameter (SQL)
SQLDataType  int(5) value;
// Length of the parameter
parmLength   int(10) value;
// Decimal number of the parameter
parmDecimal  int(5) value;
// Pointer to the buffer containing the parameter
parmBuffer   pointer value;
// Length of the buffer
bufferLength int(10);
// Length of the parameter (se alfanumerico) or 0
parmLength   int(10);
end-Pr;

//*****
// Function Prototype BIND A COLUMN TO APPLICATION VARIABLE
//
//          SQLRETURN SQLBindCol (SQLHSTMT    hstmt,
//                                 SQLSMALLINT icol,

```

```

//                                     SQLSMALLINT  fCType,
//                                     SQLPOINTER  rgbValue,
//                                     SQLINTEGER  cbValueMax,
//                                     SQLINTEGER  *pcbValue);
//*********************************************************************
// Return value = 0 (OK) or -1 (error) or -2 (invalid handle)
Dcl-pr SQLBindCol  int(10) ExtProc('SQLBindCol');
// Handle of the SQL statement
SQLHndl          int(10) value;
// Sequential parameter marker number
parmMarker        int(5)  value;
// Data type of the parameter (application)
appDataType       int(5)  value;
// Pointer to the program variable
pgmVariable       pointer value;
// Length of the variable
variableLength    int(10) value;
// Length of the parameter
parmLength        int(10);
end-Pr;

//*********************************************************************
// Function Prototype EXECUTION STATEMENT PREPARED USING SQLPREPARE
//
//          SQLRETURN SQLExecute (SQLHSTMT  hstmt);
//*********************************************************************
// Return value = 0 or 1 (OK) or 100 (no data found)
// or -1 (error) or -2 (invalid handle)
Dcl-pr SQLExecute  int(10) ExtProc('SQLExecute');
// Handle of the SQL statement
SQLHndl          int(10) value;
end-Pr;

//*********************************************************************
// Function Prototype EXECUTION DIRECT SQL STATEMENT
//
//          SQLRETURN SQLExecDirect (SQLHSTMT   hstmt,
//                                     SQLCHAR    *szSqlStr,
//                                     SQLINTEGER cbSqlStr);
//*********************************************************************
// Return value = 0 or 1 (OK) or 100 (no data found)
// or -1 (error) or -2 (invalid handle)
Dcl-pr SQLExecDir  int(10) ExtProc('SQLExecDirect');
// Handle of the SQL statement
SQLHndl          int(10) value;
// Pointer of the field containing the SQL statement
SQLStmt          pointer value;
// Length of the SQL statement
SQLStmtLength    int(5)  value;
end-Pr;

//*********************************************************************
// Function Prototype FETCH NEXT ROW
//
//          SQLRETURN SQLFetch (SQLHSTMT   hstmt)

```

```

//*****
// Return value = 0 or 1 (OK) or 100 (no data found)
// or -1 (error) or -2 (invalid handle)
Dcl-pr SQLFetch  int(10) ExtProc('SQLFetch');
// Handle of the SQL statement
SQLHndl          int(10) value;
end-Pr;

//*****
// Function Prototype LAST TRANSACTION
//
//      SQLRETURN SQLTransact (SQLHENV      henv,
//                           SQLHDBC      hdbc,
//                           SQLSMALLINT fType);
//*****
// Return value = 0 (OK) or -1 (error) or -2 (invalid handle)
Dcl-pr SQLTrans  int(10) ExtProc('SQLTransact');
// Environmental handle
envHndl          int(10) value;
// Connection handle
connHndl          int(10) value;
// Action of last transaction: 0=COMMIT, 1=ROLLBACK
action            int(5)  value;
end-Pr;

//*****
// Function Prototype RETRIEVE ERROR INFORMATION
//
//      SQLRETURN SQLError   (SQLHENV      henv,
//                           SQLHDBC      hdbc,
//                           SQLHSTMT    hstmt,
//                           SQLCHAR     *szSqlState,
//                           SQLINTEGER   *pfNativeError,
//                           SQLCHAR     *szErrorMsg,
//                           SQLSMALLINT cbErrorMsgMax,
//                           SQLSMALLINT *pcbErrorMsg);
//*****
// Return value = 0 or 1 (OK) or 100 (no data found)
// or -1 (error) or -2 (invalid handle)
Dcl-pr SQLError   int(10) ExtProc('SQLError');
// Environmental handle
envHndl          int(10) value;
// Connection handle
connHndl          int(10) value;
// Handle of the SQL statement
SQLHndl          int(10) value;
// Pointer to the field that must contain the SQLSTATE
retSQLSTATE      pointer value;
// SQLCODE returned from the database
retSQLCODE       int(10);
// Pointer to the field that must contain the error message
errorMsg         pointer value;
// Maximum length of the error message
errMaxLength    int(5)  value;
// Total length of the error message

```

```

        totalLength      int(5);
end-Pr;

//*****
// Function Prototype DEALLOCATION HANDLE OF THE SQL STATEMENT
//
//      SQLRETURN SQLFreeStmt (SQLHSTMT      hstmt,
//                                SQLSMALLINT fOption)
//*****
// Return value = 0 or 1 (OK) or -1 (error) or -2 (invalid handle)
Dcl-pr SQLFreeStm  int(10) ExtProc('SQLFreeStmt');
// Handle of the SQL statement
SQLHndl       int(10) value;
// Mode of deallocation
deallocMode    int(5)  value;
end-Pr;

//*****
// Function Prototype DISCONNECTION OF A DATABASE
//
//      SQLRETURN SQLDisconnect (SQLHDBC  hdbc);
//*****
// Return value = 0 or 1 (OK) or -1 (error) or -2 (invalid handle)
Dcl-pr SQLDisconnect  int(10) ExtProc('SQLDisconnect');
// Connection handle
connHndl       int(10) value;
end-Pr;

//*****
// Function Prototype CONNECTION DEALLOCATION HANDLE
//
//      SQLRETURN SQLFreeConnect (SQLHDBC hdbc);
//*****
// Return value = 0 (OK) or -1 (error) or -2 (invalid handle)
Dcl-pr SQLFreeCon int(10) ExtProc('SQLFreeConnect');
// Connection handle
connHndl       int(10) value;
end-Pr;

//*****
// Function Prototype DEALLOCATION ENVIRONMENTAL HANDLE
//
//      SQLRETURN SQLFreeEnv (SQLHENV  henv);
//*****
// Return value = 0 (OK) or -1 (error) or -2 (invalid handle)
Dcl-pr SQLFreeEnv  int(10) ExtProc('SQLFreeEnv');
// Environmental handle
envHndl       int(10) value;
end-Pr;

//*****
// RETCODE values
//*****
Dcl-c SQL_OK          0;

```

```

Dcl-c SQL_OK_INF      1;
Dcl-c SQL_NODATA     100;
Dcl-c SQL_NEEDAT     99;
Dcl-c SQL_ERROR      -1;
Dcl-c SQL_INVHAN    -2;

//*****
// Valid values for connect attribute
//*****
Dcl-c SQL_AUTIPD     1001;
Dcl-c SQL_ISOLVL      0;
Dcl-c SQL_NONE        1;
Dcl-c SQL_CHANGE       2;
Dcl-c SQL_CS          3;
Dcl-c SQL_ALL         4;
Dcl-c SQL_RR          5;

//*****
// SQLFreeStmt option values
//*****
Dcl-c SQL_CLOSE        0;
Dcl-c SQL_DROP         1;
Dcl-c SQL_UNBIND       2;
Dcl-c SQL_RESET        3;

//*****
// SQLTransact option values
//*****
Dcl-c SQL_COMMIT        0;
Dcl-c SQL_ROLLBK       1;
Dcl-c SQL_COMMIT_HOLD   2;
Dcl-c SQL_ROLLBK_HOLD   3;
Dcl-c SQL_SVPT_NAME_RLS 4;
Dcl-c SQL_SVPT_NAME_RB   5;

//*****
// Standard SQL data types
//*****
Dcl-c SQL_CHAR         1;
Dcl-c SQL_NUMER        2;
Dcl-c SQL_DECIM        3;
Dcl-c SQL_INTEG        4;
Dcl-c SQL_SMINT        5;
Dcl-c SQL_FLOAT        6;
Dcl-c SQL_REAL         7;
Dcl-c SQL_DOUBLE        8;
Dcl-c SQL_DATTIM       9;
Dcl-c SQL_VARCH        12;
Dcl-c SQL_BLOB         13;
Dcl-c SQL_CLOB         14;
Dcl-c SQL_DBLOB        15;
Dcl-c SQL_DATALINK     16;
Dcl-c SQL_WCHAR        17;
Dcl-c SQL_WVARCHAR     18;
Dcl-c SQL_BIGINT       19;

```

```

Dcl-c SQL_BLOB_LOCATOR 20;
Dcl-c SQL_CLOB_LOCATOR 21;
Dcl-c SQL_DBCLOB_LOC 22;
Dcl-c SQL_UTF8_CHAR 23;
Dcl-c SQL_GRAPHIC 95;
Dcl-c SQL_VARGRAPHIC 96;
Dcl-c SQL_BINARY -2;
Dcl-c SQL_VARBINARY -3;
Dcl-c SQL_DATE 91;
Dcl-c SQL_TIME 92;
Dcl-c SQL_TIMESTAMP 93;
Dcl-c SQL_CODE_DATE 1;
Dcl-c SQL_CODE_TIME 2;
Dcl-c SQL_CODD_TIMESTAMP 3;
Dcl-c SQL_ALLTYPES 0;
Dcl-c SQL_DECFLOAT -360;
Dcl-c SQL_XML -370;

//*****
// C data type to SQL data type mapping
//*****
Dcl-c SQL_C_CHAR 1;
Dcl-c SQL_C_LONG 4;
Dcl-c SQL_C_SHRT 5;
Dcl-c SQL_C_FLOT 7;
Dcl-c SQL_C_DOUB 8;
Dcl-c SQL_C_DTTM 9;

//*****
// Generally useful constants
//*****
// Null Terminated String
Dcl-c SQL_NTS -3;
Dcl-c SQL_MAXMSG 70;
Dcl-c SQL_SQLSTS 5;

```

Try it yourself: You can try this example by compiling the code from this section on your IBM i system. Change the DBName, DBUser, and DBPwd values in the SQLCLI source code to valid values for your system, and then run the following commands to create the program:

```

CRTRPGMOD MODULE(RPGISCOOL/SQLCLI) SRCFILE(RPGISCOOL/DBSRC)
CRTPGM PGM(RPGISCOOL/SQLCLI) BNDSRVPGM(QSQCLI)

```

To run the program, run the following command:

```
CALL PGM(RPGISCOOL/SQLCLI)
```

6.5 Trigger programs

A trigger is a set of actions that run automatically when a specified change operation is performed on a specified physical database file. This change operation can be an insert, update, or delete performed by an application program.

Triggers can be used for different purposes:

- ▶ Enforce business rules
- ▶ Validate input data
- ▶ Generate a unique value for a newly inserted row on a different file
- ▶ Write to other files for audit trail purposes
- ▶ Query from other files for cross-referencing purposes
- ▶ Access system functions (for example, print an exception message when a rule is violated)
- ▶ Replicate data to different files to achieve data consistency

Using triggers, customers can realize the following benefits:

- ▶ Faster application development. Because triggers are stored in the database, the actions that are performed by triggers do not have to be coded in each database application.
- ▶ Global enforcement of business rules. A trigger is defined once and then reused for any application that uses the database.
- ▶ Easier maintenance. If a business policy changes, it is necessary to change only the corresponding trigger program instead of each application program.
- ▶ Improved performance in a client/sever environment. All rules are run in the server before returning the result.

To use trigger support on the IBM i system, you must create a trigger program and add it to a physical file.

A well-modernized application and database rarely needs a trigger because the business rules are enforced by the database and changes to the rules should be isolated to a few programs or tables.

You also can create SQL triggers by running the **CREATE TRIGGER** SQL statement. SQL allows more flexibility in the types of triggers and the options for when to use a trigger. For information about how to use SQL triggers, see the DB2 for i SQL reference topic in the IBM i 7.2 Knowledge Center:

https://www.ibm.com/support/knowledgecenter/en/ssw_ibm_i_72/db2/rbafzintro.htm

6.5.1 Adding a trigger program to a file

The Add Physical File Trigger (**ADDPFTRG**) command is used to associate a trigger program with a specific physical file. After the association exists, the system calls this trigger program when a change operation is initiated against the physical file, a member of the physical file, and any logical file that is created over the physical file.

You can associate a maximum of six triggers to one physical file, one for each of the following events:

- ▶ Before an insert
- ▶ After an insert
- ▶ Before a delete
- ▶ After a delete
- ▶ Before an update
- ▶ After an update

Each insert, delete, or update event can call a trigger before the change operation occurs and after it occurs, which means that you can use the before trigger to validate the database rules. 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 operation cannot be proceeded due to an error.

To remove the association between trigger program and a file, run the Remove Physical File Trigger (**RMVPFTRG**) command. After you remove the association, no action is taken if a change is made to the physical file.

The Display File Description (**DSPFD**) command provides a list of the triggers that are associated with a file. Specify **TYPE(*TRG)** or **TYPE(*ALL)** to get this list.

6.5.2 Creating a trigger program

To provide trigger support, a trigger program must be written in a high-level language, such as SQL or CL. This example uses RPG IV.

The change operation passes two parameters to the trigger program. From these inputs, the trigger program can reference a copy of the original and new records. The trigger program must be coded to accept these parameters.

Here are the trigger program input parameters:

- ▶ Trigger buffer, which contains the information about the current change operation that started the trigger program
- ▶ Trigger buffer length

Trigger buffer field descriptions

The trigger buffer has two logical sections, which are static and variable:

- ▶ The static section contains:
 - A trigger template that contains the physical file name, member name, trigger event, trigger time, commit lock level, and CCSID of the current change record and relative record number
 - Offsets and lengths of the record areas and null byte maps
 - This section occupies the first 96 bytes
- ▶ The variable section contains:
 - Areas for the old record
 - Old null byte map
 - New record
 - New null byte map

Table 6-19 shows the content of a trigger buffer.

Table 6-19 Trigger buffer content

From	To	Type	Field description
1	10	Char(10)	Physical file name
11	20	Char(10)	Physical file library name
21	30	Char(10)	Physical file member name

From	To	Type	Field description
31	31	Char(1)	Trigger event '1'=Insert, '2'=Delete, '3'=Update
32	32	Char(1)	Trigger time '1'=After, '2'=Before
33	33	Char(1)	Commit lock level '0'=*None, '1'=*Chg, '2'=*CS, '3'=*All
34	36	Char(3)	Reserved
37	40	Integer(10)	CCSID of data
41	48	Char(8)	Reserved
49	52	Integer(10)	Original record offset
53	56	Integer(10)	Original record length
57	60	Integer(10)	Original record null byte map offset
61	64	Integer(10)	Original record null byte map length
65	68	Integer(10)	New record offset
69	72	Integer(10)	New record length
73	76	Integer(10)	New record null byte map offset
77	80	Integer(10)	New record null byte map length
81	96	Char(16)	Reserved
97	*	Char(*)	Original record
*	*	Char(*)	Original record null byte map
*	*	Char(*)	New record
*	*	Char(*)	New record null byte map

The record null byte map contains the Null value information for each field of the record. Each byte represents one field. The possible values for each byte are:

- 0 Not Null
- 1 Null

Coding snippet TRGBUF: Example of input parameters

To define input parameters for a trigger program, you can copy the source member TRGBUF from file QRPGLESRC in library QSYSINC. But, if you want to give longer and meaningful names to these fields, you must define the input data structure in your program.

Figure 6-26 illustrates how trigger program input parameters can be defined in an RPG IV program.

Example 6-26 Trigger program example

```
// Trigger buffer
Dcl-ds TrgBufferDS template qualified;          2
  FileName  char(10);
  LibName   char(10);
  MbrName   char(10);
```

```

    TrgEvent      char(1);
    TrgTime       char(1);
    CmtLevel     char(1);
    Reserved1    char(3);
    CCSID        int(10);
    Reserved2    char(8);
    OrgRecOffset int(10);
    OrgRecLength int(10);
    OrgRecNulOff int(10);
    OrgRecNullLen int(10);
    NewRecOffset int(10);
    NewRecLength int(10);
    NewRecNulOff int(10);
    NewRecNullLen int(10);
    Reserved3    char(16);
    OrgRecord    char(47);
    OrgRecordNul char(5);
    NewRecord    char(47);
    NewRecordNul char(5);

    // Overlay fields for original record
    PartNumOrg   zoned(5 : 0) Overlay(OrgRecord);          4
    PartDesOrg   char(25)    Overlay(OrgRecord:*Next);
    PartQtyOrg   packed(5 : 0) Overlay(OrgRecord:*Next);
    PartPrcOrg   packed(6 : 2) Overlay(OrgRecord:*Next);
    PartDatOrg   date       Overlay(OrgRecord:*Next);

    // Overlay fields for new record
    PartNumNew   zoned(5 : 0) Overlay(NewRecord);          4
    PartDesNew   char(25)    Overlay(NewRecord:*Next);
    PartQtyNew   packed(5 : 0) Overlay(NewRecord:*Next);
    PartPrcNew   packed(6 : 2) Overlay(NewRecord:*Next);
    PartDatNew   date       Overlay(NewRecord:*Next);
    end-Ds;

    // Entry parameter list for trigger program
    DCL-PI partsTrigger;                                1
    trigBuffer likeds(trgBufferDS);
    trigBufLen int(10) const;                            3
    END-PI ;
    Date=%Subst(trigBuffer: trigbuffer.OrgRecOffset + 38 : 10);  5

```

TRGBUF snippet notes

The notes in this section refer to the numbers in the right side of Example 6-26 on page 239:

1. The entry parameter list defines two parameters: the trigger buffer and trigger buffer length.
2. The trigger buffer is defined as a data structure.
3. The trigger buffer length is a stand-alone field, which is type integer.
4. Using the Overlay keyword, you can redefine original and new record fields in the trigger buffer to access particular fields from those records.
5. Record offset in combination with the %SUBST built-in function can be used to access particular fields from a trigger buffer.

Trigger program coding guidelines

Observe these guidelines for trigger program coding:

- ▶ The trigger program is called for each row that is changed in the physical file.
- ▶ The trigger program and application program may run in the same or different activation groups. As a preferred practice, compile the trigger program with **ACTGRP(*CALLER)** to achieve consistency between the trigger program and the application program.
- ▶ A commit lock level of the application program is passed to the trigger program. As a preferred practice, have the trigger program run under the same lock level as the application program.
- ▶ A trigger program can call other programs or can be nested so that a statement in a trigger program causes the calling of another trigger program. In addition, a trigger program may be called recursively by itself. The maximum trigger nested level for insert and update is 200.
- ▶ When the trigger program runs under commitment control, the following situations result in an error:
 - Any update of the same record that was already changed by the change operation or by an operation in the trigger program.
 - Conflicting operations on the same record within one change operation. For example, a record is inserted by the change operation and then deleted by the trigger program.
- ▶ The Allow Repeated Change **ALWREPCHG(*YES)** parameter on the Add Physical File Trigger (**ADDPFTRG**) command also affects trigger programs that are defined to be called before insert and update database operations. If the trigger program updates the new record in the trigger buffer and **ALWREPCHG(*YES)** is specified, the modified new record image is used for the actual insert or update operation on the associated physical file. This option can be helpful in trigger programs that are designed for data validation and data correction.

Trigger program error messages

If a failure occurs while the trigger program is running, it should send an appropriate escape message before exiting. Otherwise, the application assumes the trigger program ran successfully. The message can be the original message received from the system or a message created by the programmer.

6.6 Commitment control

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. 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.

The typical example of changes that can be grouped together is the transfer of funds from a savings to a 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 LUWs from a prior failure.

6.6.1 File journaling

Commitment control requires that a database file or table must be journaled before it can be opened for output under commitment control. A file does not need to be journaled to open it for input only under commitment control.

Commitment control uses a journal to write entries that identify the beginning and ending of commitment control, the start of a commit cycle, or a commit and rollback operation. It also uses the before record images when a rollback operation is performed.

If only the after images are being journaled for a database file, when that file is opened under commitment control, the system automatically starts journaling both the before and after images. The before images are written only for changes to the file that occur under commitment control.

When using SQL, a journal and journal receiver are automatically created as a part of the SQL collection. Any SQL table that is created inside the SQL collection is automatically journaled.

If the database file is created by using a native interface (CRTPF), you must take care to activate journaling. Three steps are required:

1. Run the Create Journal Receiver (**CRTJRNRCV**) command to create a journal receiver object.
2. Run the Create Journal (**CRTJRN**) command to create a journal object and associate it with the journal receiver.
3. Use the Start Journal Physical File (**STRJRNPF**) command to start journaling for selected files.

6.6.2 Using commitment control with RPG native file operations

To use commitment control in an ILE RPG program with native file operations, complete the following steps:

1. On the system, complete the following steps:
 - a. Prepare for using commitment control to start journaling for all files that will be used under commitment control.
 - b. Notify the system when to start and end commitment control. Use the CL commands Start Commitment Control (**STRCMCTL**) and End Commitment Control (**ENDCMCTL**).
2. In the ILE RPG program, complete the following steps:
 - a. Specify commitment control (COMMIT) on the file-description specifications of the files you want under commitment control.
 - b. Use the COMMIT operation code to apply a group of changes to files under commitment control, or use the ROLBK (Roll Back) operation code to eliminate the pending group of changes to files under commitment control in the current transaction.

Starting commitment control

With the CL command Start Commitment Control (**STRCMTCTL**), you notify the system that you want the commitment control to be started with the following options:

- ▶ The Lock level parameter (**LCKLVL**) defines the level at which records are locked under commitment control:
 - ***CHG** Every record read for an update is locked. If a record is changed, added, or deleted, that record remains locked until the transaction is committed or rolled back. Records that are accessed for update operations, but are released without being changed, are unlocked.
 - ***CS** Every record accessed is locked. A record that is read, but not changed or deleted, is unlocked when a different record is read. Records that are changed, added, or deleted are locked until the transaction is committed or rolled back.
 - ***ALL** Every record accessed is locked until the transaction is committed or rolled back.
- ▶ The Commitment definition scope parameter (**CMTSCOPE**) specifies the scope for the commitment definition, either the activation group level or the job level.
- ▶ A Notify object specifies the name and type of object (file, data area, or message queue) where notification is sent in the event of an abnormal job end. This information relates to the last successfully completed group of changes and is used to restart the job.

Commit and rollback operations

To indicate that a database file should run under commitment control, enter the keyword **COMMIT** in the keyword field of the file description specification.

When a program specifies commitment control for a file, the specification applies only to the input and output operations that are made by this program for this file. Other programs that use the same file without commitment control are not affected.

The **COMMIT** keyword has an optional parameter, which allows conditional use of commitment control. The ILE RPG compiler implicitly defines a one-byte character field with the same name as the one specified as the parameter. If the parameter is set to 1, the file runs under commitment control.

The **COMMIT** keyword parameter must be set before opening the file. You can do so by passing in a value when you call the program or by explicitly setting it to "1" in the program.

On a calculation line, you can use two operation codes at the end of a related group of changes to denote the end of a transaction:

- ▶ The **COMMIT** operation tells the system that a group of changes to the files under commitment control is completed.
- ▶ The **ROLBK** operation eliminates the current group of changes to the files under commitment control.

If the system fails, it implicitly issues a **ROLBK** operation. You can check the identity of the last successfully completed group of changes by using the label you specify in factor 1 of the **COMMIT** operation code and the notify object you specify on the Start Commitment Control (**STRCMTCTL**) command.

Example 6-27 illustrates the use of commitment control in a program with native file operations.

Example 6-27 Commit control in a program with native file operations

```

dcl-f Parts disk(*ext) usage(*update) keyed Commit(ComitFlag);
dcl-f Trans disk(*ext) usage(*update) keyed Commit(ComitFlag);

// If ComitFlag = '1' the files are opened under commitment control,
// otherwise they are not.

DCL-PI cmtExample;
    ComitFlag ind;
END-PI ;

// Use the COMMIT operation to complete a group of operations if
// they were successful or rollback the changes if they were not
// successful. You only issue the COMIT or ROLBK if the files
// were opened for commitment control (ie. COMITFLAG = '1')

Update(E) PartR;
Update(E) TranR;

If ComitFlag = *on;

If %Error;
    Rolbk;
Else;
    Commit;
EndIf;

EndIf;                                // end if commitment control

```

6.6.3 Using commitment control with embedded SQL

SQL provides similar functions to support commitment control by using the embedded SQL statements **COMMIT** and **ROLLBACK**.

Starting commitment control

The commitment control environment for ILE RPG programs with embedded SQL statements is started automatically. DB2 for IBM i implicitly calls the CL command Start Commitment Control (**STRCMTCTL**) and specifies the requested parameters **NFYOBJ(*NONE)** and **CMTSCOPE(*ACTGRP)**. The **LCKLVL** parameter is specified according to the lock level on the **COMMIT** parameter of the Create SQL ILE RPG Object (**CRTSQLRPGI**) command, which is used to create the program.

SQL supports the following lock levels:

- *NONE or *NC Specifies that commitment control is not used. Uncommitted changes in other jobs can be seen.
- *CHG or *UR The updated, deleted, and inserted rows are locked until the end of the transaction. Uncommitted changes in other jobs can be seen.

*CS	The updated, deleted, and inserted rows are locked until the end of the transaction. A row that is selected, but not updated, is locked until the next row is selected. Uncommitted changes in other jobs cannot be seen.
*ALL or *RS	The selected, updated, deleted, and inserted rows are locked until the end of the transaction. Uncommitted changes in other jobs cannot be seen.
*RR	The selected, updated, deleted, and inserted rows are locked until the end of the transaction. Uncommitted changes in other jobs cannot be seen. All tables referred to in SELECT , UPDATE , DELETE , and INSERT statements are locked exclusively until the end of the transaction.

Commit and rollback statements

In the program, you can either commit all transaction changes by using the embedded SQL statement **COMMIT** or back out of these changes by using the **ROLLBACK** statement.

Both statements have an optional **HOLD** parameter, which can be useful when using the cursor technique to fetch rows from a result table. If **HOLD** is specified, currently open cursors are not closed, and all resources that are acquired during the unit of work are held. Locks on specific rows and objects that are implicitly acquired during the unit of work are released.

If **HOLD** is omitted, cursors opened within this unit of work are closed unless the cursors were declared with the **WITH HOLD** clause.

6.6.4 Using commitment control with the CLI interface

The CLI interface provides support for commitment control by calling specific CLI functions, as described earlier in 6.4, “DB2 call level interface” on page 200.

Starting commitment control

The function **SQLSetConnectionOption()** sets connection attributes, including the transaction isolation level for commitment control. This function should be run in the initialization phase of the example CLI program to define the requested level of commitment control.

The following connect options are supported:

- ▶ **SQL_COMMIT_NONE**
- ▶ **SQL_COMMIT_CHG**
- ▶ **SQL_COMMIT_CS**
- ▶ **SQL_COMMIT_ALL**
- ▶ **SQL_COMMIT_RR**

Example 6-28 contains an example of calling **SQLSetConnectionOption()** to define the change isolation level for commitment control.

Example 6-28 Example using SQLSetConnectionOption()

```
// Variable definitions
Dcl-s ConnOpt  int(5);
Dcl-s Is1Lvl   int(10);
Dcl-s Is1LvlP  pointer INZ(%ADDR(Is1Lvl));

// Set connection options
ConnOpt=SQL_ATTR_COMMIT;
Is1Lvl=SQL_COMMIT_CHG;
SQL_RC=SQLSetCnOp(hdbc:ConnOpt:Is1LvlP);
```

SQL_ATTR_COMMIT is a numeric constant with a value of 0 and signals to CLI that you want to define an option for commitment control.

SQL_COMMIT_CHG is another numeric constant with a value of 2 and defines the change isolation level.

The commit and rollback function

Another function, **SQLTransact()**, should be used to perform commit or rollback operations at the end of a transaction. All changes to the database that are performed on the connection since connect time or the previous call to **SQLTransact()** are committed or rolled back.

The following code snippet shows how to start this function:

```
SQL_RC=SQLTrans(henv:hdcb:SQL_COMMIT);
```

The numeric constant SQL_COMMIT with value 0 defines a commit operation. Another constant SQL_ROLLBACK, with value 1, represents a rollback operation.

6.7 A note about globalization

With the web becoming a more common interface and businesses becoming global, storing and managing data in multiple languages is important. The web uses Unicode to send and receive data. IBM i supports Unicode in the database. ILE RPG continues to improve the conversion between other character code pages (CCSIDs) and Unicode.

For more information about the globalization of your application, see the IBM i globalization topic in the IBM i 7.2 Knowledge Center:

http://www.ibm.com/support/knowledgecenter/en/ssw_ibm_i_72/nls/rbagsglobalmain.htm

6.8 More information about database access with RPG IV

Where should you go for more information and samples of RPG IV applications integrating with DB2 for IBM i?

Consider the following IBM manuals for reference that are available on the IBM i 7.2 Knowledge Center:

https://www.ibm.com/support/knowledgecenter/en/ssw_ibm_i_72/rzahg/rzahgdref.htm

- ▶ DB2 for i SQL Programming
- ▶ DB2 for i SQL Reference
- ▶ DB2 for i SQL Call Level Interface

For programming samples and frequently asked questions (FAQs) and their answers, see the following websites:

- ▶ DB2 for i Frequently Asked Questions

<http://www-01.ibm.com/common/ssi/cgi-bin/ssialias?subtype=ST&infotype=SA&htmlfid=POQ12365USEN&attachment=POQ12365USEN.PDF>

- ▶ IBM DB2 for i: Tips, SQL CLI frequently asked questions:

<http://www.ibm.com/systems/power/software/i/db2/support/tips/clifaq.html>

- ▶ IBM DB2 for i tips and techniques:
<http://www.ibm.com/systems/power/software/i/db2/support/tips/>
- ▶ IBM DB2 for i website:
<http://www.ibm.com/systems/power/software/i/db2/>



Exception and error handling

The original version of this book briefly covered the topic of exception and error handling, mostly from the perspective of ILE exception handlers. Since that time, a number of things have happened:

- ▶ Interest in error handling has grown among RPG programmers, particularly with the increased use of subprocedures.
- ▶ New error handling techniques were introduced into the RPG language itself through the MONITOR op-code.

For these reasons, this chapter handles the topic of exception and error handling in a more comprehensive manner.

This chapter was originally published as an IBM Redpaper™ publication. The code samples are updated for free-form RPG and a section about placing embedded SQL for producing exceptions and errors is included. This chapter provides an overview of the traditional OPM style methods of implementing exception/error handling and how they change in the ILE environment. It also describes how you can use “pure” ILE methods to replace some of the more traditional methods.

The following exception and error handling topics are discussed in this chapter:

- ▶ 7.1, “Introduction to exception and error handling” on page 250
- ▶ 7.2, “What is an exception/error” on page 250
- ▶ 7.3, “Trapping at the program level” on page 251
- ▶ 7.4, “Trapping at the operation level” on page 255
- ▶ 7.5, “Subprocedures and exception/errors” on page 266
- ▶ 7.6, “ILE CEE APIs” on page 287
- ▶ 7.7, “Priority of handlers” on page 298
- ▶ 7.8, “The embedded SQL problem” on page 299
- ▶ 7.9, “Using percolation: try, throw, and catch” on page 302
- ▶ 7.10, “Conclusion” on page 309

7.1 Introduction to exception and error handling

Exception/error handling is the term given to handling unexpected exceptions/errors in programs. No matter how well you test your programs, there is always the possibility that an unforeseen error may occur, for example, a divide by zero, an array index error, or a duplicate record on a write to a file.

The most common indication that you have an unexpected exception/error is that a program fails with a CPF message. This situation can be disastrous if the program fails in a production environment and a user receives the CPF message; they either have a protracted conversation with the help desk or, even worse, they simply press enter and, by default, cancel the program.

Users should never, ever see the “two-line screen of death” that is caused by an unhandled CPF message. Exception/error handling provides all the tools that you need to ensure that, should a program fail, it terminates in an orderly fashion, displays a friendly message to the user, notifies the help desk that there is a problem, and provides enough documentation for a programmer to determine what caused the problem.

There is also the case where you might want to trap exception/errors for a specific operation or group of operations and handle them in the program. This situation applies especially to file operations where it is quite reasonable for a program to handle a duplicate record condition on a WRITE operation or a constraint violation on a DELETE operation.

7.2 What is an exception/error

Your program makes a request (an RPG operation) to the operating system. The request fails and the operating system responds by sending an escape message to the program message queue. When the escape message arrives, a check is made to see whether the program is going to handle the error (`MONMSG` in CL) and, if not, default error handling kicks in. You should be aware that there is a fundamental difference between default error handling for OPM and for ILE. OPM default error handling means that a CPF message is immediately sent to indicate that the program failed, but for ILE, the error message is first “percolated” back up the call stack to see if the calling procedure is going to handle the error. This process continues until the boundary of the activation group is reached. You will see the difference between OPM style and ILE style exception/error handling as you progress through the chapter.

If you do not want the RPG default error handler to handle exception/errors, RPG provides a number of alternative methods:

- ▶ Trap exception/errors for a complete program by using a *PSSR subroutine
- ▶ Trap exception/errors for individual files by using an INFSR subroutine
- ▶ Trap exception/errors for a single operation by using an E extender
- ▶ Trap exception/errors for a group of operations by using a MONITOR group

Throughout this chapter, you will see how all of these alternative methods are handled, starting with handling unexpected exception/errors at a program level. The examples illustrate a simple method to provide consistently an orderly shutdown of the program.

7.3 Trapping at the program level

Start with the traditional methods of handling exception/errors in RPG programs that run in OPM style (that is, that run in the default activation group). In 7.5, “Subprocedures and exception/errors” on page 266, you see how most of these traditional methods may be replaced in an ILE environment.

RPG differentiates between program exception/errors (divide by zero, array index error, and so on) and file exception/errors (record lock, duplicate record, and so on).

The sample program ERROR01 that is shown in Example 7-1 demonstrates two conditions that can cause an exception/error.

Example 7-1 Example of two exception/error conditions

```
dcl-F product1 disk(*ext) usage(*update: *delete) keyed usr0pn;

dcl-S dummyCode like(prodCd);
dcl-S a          int(10) inz(20);
dcl-S b          int(10);
dcl-S c          int(10);

c = a/b;

chain dummyCode product1;
open product1;

*inLR = *on;

// Indi ind execWhen
```

Note: You can create this program by running the following command:

```
CRTBNDRPG PGM(REDBOOK/ERROR01) SRCFILE(REDBOOK/EXCEPTSRC)
```

The first exception/error is on the divide operation, which results in an RNQ0102 Attempt to divide by zero message being issued because b has a value of 0.

The second exception/error occurs on the **CHAIN** operation, which because an attempt is made to retrieve a record before opening the file, results in an RNQ1211 I/O operation was applied to closed file PRODUCT1 message being issued.

Of course, when you run this program, you see only the divide by zero error because once the error occurs, you are presented only with options to cancel the program, continue at *GETIN (that is, the start of the mainline), or obtain a dump.

7.3.1 Program exception/errors

If a program encounters an otherwise unhandled program exception/error, it checks to see whether there is a *PSSR subroutine that is coded in the program. If there is, it runs the subroutine; if not, the program “fails” with an error message.

ERROR02, which is shown in Example 7-2, is the same sample program as ERROR01 but with a *PSSR subroutine included.

*Example 7-2 Example ERROR02 with a *PSSR subroutine*

```
dcl-F product1 disk(*ext) usage(*update: *delete) keyed usr0pn;

          dcl-Pr fatalProgram extPgm('FATALPGM');
          end-Pr;

(1)   dcl-S PSSRDone ind;
          dcl-S dummyCode like(prodCd);
          dcl-S a           int(10) inz(20);
          dcl-S b           int(10);
          dcl-S c           int(10);

          c = a/b;

          chain dummyCode product1;
          open product1;

          *inLR = *On;

          begSR *PSSR;
(2)     dump(A);
(3)     if not PSSRDone;
          PSSRDone = *On;
          fatalProgram();
          endIf;
(4)     endSR '*CANCL';
```

Note: You can create this program by running the following command:

```
CRTBNDRPG PGM(REDBOOK/ERROR02) SRCFILE(REDBOOK/EXCEPTSRC)
```

Later, you see how the *PSSR routine can be placed in a copy member but, for now, here are the main points to note about the inclusion of the *PSSR routine. Note the numbers on the left side of Example 7-2, correspond to the following items:

1. The PSSRDone indicator and the prototype for FatalProgram usually are included in a standard copy member as with prototypes and standard field definitions. The code for FatalProgram is shown below in Example 7-4 on page 253.
2. The (A) extender on the DUMP operation means that the dump A (always) takes place, regardless of whether the program was compiled with the **DEBUG** option.
3. The reason for the PSSRDone indicator is in case there is an exception/error in the *PSSR subroutine itself, which causes the *PSSR subroutine to be run, which fails and causes the *PSSR subroutine to be run, which fails until the number of dump listings cause a serious problem on the system.
4. One of the issues with a *PSSR is that when it is invoked by an error, it does not provide an option to return to the statement in error. Therefore, it should be used as a means of performing an orderly exit from the program. In theory, the EndSR for a *PSSR subroutine can contain a return-point instruction, but most of them apply only if you use the RPG cycle (which should never be used anymore), and even then they are of marginal to dubious utility. Here, the value of *CANCL indicates that the program should cancel when it

reaches the end of the subroutine, but, as you see, this should never happen because the fatal program never returns from the call.

Now, the divide by zero error results in the *PSSR subroutine being run as opposed to the RNQ0102 message being issued.

7.3.2 The fatal program

FATALDSPF (Example 7-3) is a display file that is used to display a meaningful message to the user if an interactive job fails.

Example 7-3 FAATALDSP display file details

```
A          R FATALERROR
A
A          10 21'Oh Dear. It has all gone terribly -
A          wrong!
A          COLOR(RED)
```

FATALPGM (Example 7-4) is a CL program that gets called when a program fails with an unexpected exception/error.

Example 7-4 FTALPGM source code for CL program

```
DclF RedBook/FatalDsp
Dcl  &Job      *Char (10)
Dcl  &User     *Char (10)
Dcl  &Nbr      *Char (6)
Dcl  &Type     *Char (1)
Dcl  &Reply    *Char (1)

DspJobLog Output(*Print)
SndMsg ('It has all just gone horribly wrong!') ToMsgQ(QSysOpr)

RtvJobA Job(&Job) User(&User) Nbr(&Nbr) Type(&Type)
ReShow:
If (&Type = '1') SndRcvF RcdFmt(FATALERROR)
Else SndUsrMsg Msg('You must end Job ' *Cat &Nbr *TCat '/' +
                    *Cat &User *TCat '/' *CAT &Job *BCat +
                    'now. Error reports have been produced') +
                    ToMsgQ(*Sysopr) MsgRpy(&Reply)
Goto ReShow
```

Note: You can create this program by running the following commands:

```
CRTDSPF FILE(REDBOOK/FATALDSP) SRCFILE(REDBOOK/EXCEPTSRC)
CRTBNDCL PGM(REDBOOK/FATALPGM) SRCFILE(REDBOOK/EXCEPTSRC)
```

The program prints a job log, sends a message to the system operator, and either displays a meaningful/helpful screen for the user (if running interactively) or sends an inquiry message to the system operator (if running in batch). The screen is displayed or the message is sent continuously until the job is cancelled.

The complexity of the program called from the *PSSR subroutine is up to you. It does not have to be a CL program; it could be another RPG program. You might decide to have the program perform some sort of logging. You might pass an information data structure or status codes as parameters. It all depends on the level of detail you want to go to.

You decide what you want to put on the screen format FATALERROR in the display file FATALDSP; just keep it friendly and non-threatening. It is often useful for your screen to include an entry area where you can solicit information from the user about what they were doing at the time. Were they doing anything different or unusual? Did anything else odd happen today?

But with the changes suggested so far, if a program fails in a production environment, the user is looking at a helpful screen and you have a program dump and a job log to help identify the problem.

You see another example of the type of functions that can be performed by a program in 7.5.3, “Identifying a percolated message” on page 272.

7.3.3 File exception/errors

You now change the sample program so that it no longer fails with a decimal data error by simply specifying a non-zero value for b. But now, when you run the program, it fails with an RNQ1211 I/O operation was applied to closed file PRODUCT1 message. The automatic execution of the *PSSR routine does not apply to file errors in the same way that it applies to program errors.

Just as a program has a *PSSR subroutine to process unhandled program exception/errors, each file can have its own subroutine to handle file exception/errors. This subroutine is identified by using the **INFSR** keyword on the F spec for the file. This method works for a traditional style program, but does not work if a file is referenced in a subprocedure. You see how to handle file exceptions/errors in subprocedures in 7.5.2, “Trapping percolated errors” on page 271.

Using a separate subroutine per file made a lot of sense when using the RPG cycle, but is of little use within a full procedural environment. Each subroutine simply must provide an orderly means of exiting the program and you already have that in your *PSSR routine. Therefore, one way of handling file errors is to simply identify the *PSSR routine on the **INFSR** keyword for each file in your program, as shown by the line with a “1” next to it in the program ERROR03 shown in Example 7-5.

Example 7-5 Source code for example ERROR03

```

dcl-F product1 disk(*ext) usage(*update: *delete)
(1)          keyed usrOpn infSR(*PSSR);

dcl-Pr fatalProgram extPgm('FATALPGM');
end-Pr;

dcl-S PSSRDone ind;
dcl-S dummyCode like(prodCd);
dcl-S a      int(10) inz(20);
dcl-S b      int(10) inz(10);
dcl-S c      int(10);

c = a/b;

chain dummyCode product1;
open product1;

*inLR = *On;

```

```

begSR *PSSR;
dump(A);
if not PSSRDone;
PSSRDone = *0n;
fatalProgram();
endIf;
endSR '*CANCL';

```

Note: You can create this program by running the following command:

```
CRTBNDRPG PGM(REDBOOK/ERROR03) SRCFILE(REDBOOK/EXCEPTSRC)
```

Now, with one exception, if there is an exception/error on a file operation, the *PSSR subroutine is run.

The one exception is with the opening of files. Normally, the files are automatically opened by the RPG cycle. But, this process occurs before any user code is started, therefore it cannot run *PSSR if there is a problem during file open (for example, level check).

To trap file open errors, you must perform an implicit open of the file by specifying a conditional open for the file (**USR0PN** keyword on F Spec) and use the OPEN operation to open the file (usually coded in the *INZSR subroutine).

7.3.4 Can it get any easier

Because the *PSSR routine that is suggested is the same in every program, why not code it in its own member and include it in any program by using a /**COPY** directive? In further program examples, the *PSSR subroutine is placed in a member named PSSRSTD. Of course, this means that it is much simpler to enhance your error handling procedures because changing this one piece of code enhances all of the programs that use it.

The coding requirement for basic exception/error handling is to define an **INFSR(*PSSR)** for every file and a copy directive to include the *PSSR subroutine.

In 7.5.2, “Trapping percolated errors” on page 271, you see how a *PSSR routine needs to be coded only in one program when programs are running in an ILE environment.

7.4 Trapping at the operation level

One program’s error is another program’s feature. Certain exception/errors might be expected in a program, in which case you might not want the *PSSR routine to be invoked. For example, you might want your program to trap and handle exceptions for record locking, constraint violations, and certain decimal data errors.

Exception/errors may be trapped at the operation level by using error extenders or MONITOR groups.

7.4.1 Error extender

Some operation codes offer the equivalent of a command level **MONMSG** in CL. This is implemented through the error extender (replacing the old error indicator in the low position) and allows the programmer to respond to errors instead of the operation failing. Instead, if it receives an error, the %ERROR BIF is set and processing continues with the next operation. It is

up to the programmer to determine whether there was an exception/error and what to do about it.

Program ERROR04 shown in Example 7-6 demonstrates the use of the error extender.

Example 7-6 Source code for example ERROR04

```
dcl-F product1 disk(*ext) usage(*update: *delete) keyed usr0pn;

dcl-Pr fatalProgram extPgm('FATALPGM');
end-Pr;

dcl-S PSSRDone ind;
dcl-S dummyCode like(prodCd);
dcl-S a      int(10) inz(20);
dcl-S b      int(10);
dcl-S c      int(10);

c = a/b;

(1)   chain(E) dummyCode product1;
(2)   if %error();
      dsply 'The file has not been opened';
endif;

(3)   chain dummyCode product1;
open product1;

*inLR = *On;

/include EXCEPTSRC,PSSRSTD
```

Note: You can create this program by running the following CL command:

```
CRTBNDRPG PGM(REDBOOK/ERROR04) SRCFILE(REDBOOK/EXCEPTSRC)
```

Note the following points in Example 7-6 as indicated by the numbers on the left side of the example:

1. The error extender on the CHAIN operation means that the programs continues to run with the next operation as opposed to running the subroutine that is specified on the INFSR (that is, *PSSR) or failing with an error message.
2. You check for an error by using the %ERROR BIF. In this example, the DSPLY operation is used to display a message if there was an exception/error on the CHAIN operation.
3. Processing continues with the next CHAIN operation which, as before, results in the running of the *PSSR subroutine.

Using the error extender does not apply to all operation codes, but it does apply to all file and "external related" operation codes (CALL, IN, and OUT). See 7.4.3, "Monitor groups" on page 260 to see how you can use MONITOR for the other operation codes.

Table 7-1 lists the operation codes that are eligible for an error extender or error indicator.

Table 7-1 Operation codes that are eligible for an error extender

ACQ	CLOSE	OCCUR	REL	TEST
-----	-------	-------	-----	------

ADDUR	COMMIT	OPEN	RESET	UNLOCK
ALLOC	DEALLOC	NEXT	ROLBK	UPDATE
CALL	DELETE	OUT	REALLOC	WRITE
CALLB	DSPLY	POST	SCAN	XLATE
CALLP	EXFMT	READ	SETGT	XML-INTO
CHAIN	EXTRCT	RAEDE	SETLL	XML-SAX
CHECK	FEOD	READP	SUBDUR	
CHECKR	IN	READPE	SUBST	

7.4.2 Status codes

How do you know which exception/error has occurred? The %STATUS BIF provides a five-digit status code that identifies the error. Program status codes are 00100 - 00999 and file status codes are 01000 - 01999. Status codes 00000 - 00050 are considered normal (that is, they are not set by an exception/error condition).

Status codes correspond to RPG runtime messages in the message file QRNXMSG (for example, Message RNQ0100 = Status Code 00100). You can view the messages by running the following command:

```
DSPMSGD RANGE(*FIRST *LAST) MSGF(QRNXMSG) DETAIL(*FULL)
```

Table 7-2 lists some of the more commonly used Status Codes.

For a full list of status codes, see the *RPG IV File and Program Exception/Errors* topic in the IBM i 7.2 Knowledge Center:

http://www.ibm.com/support/knowledgecenter/en/ssw_ibm_i_72/rzasd/f1pgxpt.htm#f1pgxpt

Table 7-2 Commonly used status codes

Code	Description
00100	Value out of range for string operation.
00102	Divide by zero.
00112	Invalid date, time, or timestamp value.
00121	Array index not valid.
00122	OCCUR outside of range.
00202	Called program or procedure failed.
00211	Error calling program or procedure.
00222	Pointer or parameter error.
00401	Data area specified on IN/OUT not found.
00413	Error on IN/OUT operation.
00414	User not authorized to use data area.

00415	User not authorized to change data area.
00907	Decimal data error (digit or sign not valid).
01021	Tried to write a record that already exists (file being used has unique keys and key is duplicate, or attempted to write duplicate relative record number to a subfile).
01022	Referential constraint error detected on file member.
01023	Error in trigger program before file operation is performed.
01024	Error in trigger program after file operation is performed.
01211	File not open.
01218	Record already locked.
01221	Update operation attempted without a prior read.
01222	Record cannot be allocated due to referential constraint error.
01331	Wait time exceeded for READ from WORKSTN file.

It is useful to define a copy member that contains the definition of named constants that correspond to the status codes. This provides for “self-documenting” code. For example, the copy member ERRSTATUS contains the definitions that are shown in Example 7-7 that correspond to the status codes that are shown in Table 7-2 on page 257.

Example 7-7 Source code for ERRSTATUS

```
dcl-C ERR_VALUE_OUT_OF_RANGE      00100;
dcl-C ERR_DIVIDE_BY_ZERO         00102;
dcl-C ERR_INVALID_DATE          00112;
dcl-C ERR_ARRAY_INDEX           00121;
dcl-C ERR_OCCUR_OUT_OF_RANGE    00122;
dcl-C ERR_CALL_FAILED           00202;
dcl-C ERR_CALLING               00211;
dcl-C ERR_POINTER_OR_PARAMETER  00222;
dcl-C ERR_DATA_AREA_NOT_FOUND   00401;
dcl-C ERR_IN_OR_OUT              00413;
dcl-C ERR_USE_DATA_AREA          00414;
dcl-C ERR_CHANGE_DATA_AREA       00415;
dcl-C ERR_DECIMAL_DATA           00907;
dcl-C ERR_DUPLICATE_WRITE        01021;
dcl-C ERR_REFERENTIAL_CONSTRAINT 01022;
dcl-C ERR_TRIGGER_BEFORE         01023;
dcl-C ERR_TRIGGER_AFTER          01024;
dcl-C ERR_NOT_OPEN                01211;
dcl-C ERR_RECORD_LOCKED          01218;
dcl-C ERR_UPDATE_WITHOUT_READ    01221;
dcl-C ERR_CANNOT_ALLOCATE_DUE_TO_RC 01222;
dcl-C ERR_WAIT_EXCEEDED          01331;
```

The program ERROR04 simply assumed that the error being trapped was for the status 1211 (File not open). Program ERROR05, shown in Example 7-8, shows how to ensure that only a status of 1211 is being trapped.

Example 7-8 Source code for program ERROR05

```
dcl-F product1 disk(*ext) usage(*update: *delete) keyed usr0pn;

dcl-Pr fatalProgram extPgm('FATALPGM');
end-Pr;

dcl-S PSSRDone ind;
dcl-S dummyCode like(prodCd);
dcl-S a          int(10) inz(20);
dcl-S b          int(10);
dcl-S c          int(10);

(1) /include EXCEPTSRC,ERRSTATUS

c = a/b;

chain(E) dummyCode product1;
if %error();
(2)   if %status(product1) = ERR_NOT_OPEN;
        dspl 'The file has not been opened';
(3)   else;
        exSR *PSSR;
    endIf;
endIf;

chain dummyCode product1;
open product1;

*inLR = *On;

/include EXCEPTSRC,PSSRSTD
```

Note: You can create this program by running the following command:

```
CRTBNDRPG PGM(REDBOOK/ERROR05) SRCFILE(REDBOOK/EXCEPTSRC)
```

Note the following points in Example 7-8 as indicated by the numbers on the left side of the example:

1. A **/COPY** directive includes the copy member (ERRSTATUS) containing the named constants for the status codes.
2. The IF condition for the **%ERROR** BIF now uses the **%STATUS** BIF to check whether the status of the file indicates a file not open condition. You should always specify the file name with the **%STATUS** BIF to enable you to differentiate between the current statuses of different files. It is a preferred practice to do this even if you only have one file coded in the program because you might need other files later.
3. Otherwise, the program falls back on the ***PSSR** routine.

7.4.3 Monitor groups

But what about those operation codes that do not allow an error extender (as discussed in 7.4.1, “Error extender” on page 255)? Is it possible to trap exception/errors for them on an individual basis? Yes it is.

RPG contains a MONITOR group, which you can use to monitor a number of statements for potential errors, as opposed to checking them one at a time. In concept, it is quite similar to a SELECT group. It has the following structure:

```
monitor;
    // Code to monitor
on-Error statuscode : statuscode :statuscode;
    // Handle Error
on-Error *FILE;
    // Handle Error
on-Error *PROGRAM;
    // Handle Error
on-Error *ALL;
    // Handle Error
endMon;
```

A MONITOR group works as follows:

- ▶ The boundary is set by a MONITOR and an ENDMON operation.
- ▶ The MONITOR operation is followed by the code that you want to monitor for a possible exception/error, which can be a single line of code, an entire program, or anything in between. It also can include other monitor blocks.
- ▶ The code is followed by a set of ON-ERROR operations that indicate the status codes to trap along with the appropriate code to handle the specified error. You may specify individual status codes or the reserved values of *FILE (any file exception/error), *PROGRAM (any program exception/error), or *ALL (any file or program exception/error).
- ▶ If no error occurs in the MONITOR block (that is, before the first ON-ERROR operation is reached), processing continues at the first instruction following the ENDMON operation code.
- ▶ If an error is detected in the block, control immediately passes to the ON-ERROR operations which are processed in sequence, working in the same way as WHEN operations in a SELECT group.
- ▶ If a match is found, the code for the ON-ERROR segment is processed and control passes to the instruction following the ENDMON operation. If no matching ON-ERROR condition is found, normal default handling applies (as thought the MONITOR group did not exist).
- ▶ Monitor groups may be nested. For some reason, this seems to be a feature that is under utilized by many RPG programmers.

Program ERROR06 (Example 7-9) shows the implementation of a MONITOR group.

Example 7-9 Source code for program ERROR06

```
dcl-F product1 disk(*ext) usage(*update: *delete) keyed usr0pn;

dcl-Pr fatalProgram extPgm('FATALPGM');
end-Pr;

dcl-S PSSRDone ind;
dcl-S dummyCode like(prodCd);
```

```

dcl-S a      int(10) inz(20);
dcl-S b      int(10);
dcl-S c      int(10);

/include EXCEPTSRC,ERRSTATUS

(1)  monitor;
      c = a/b;

(2)  chain dummyCode product1;

      chain dummyCode product1;
      open product1;

(3)  on-Error ERR_NOT_OPEN;
      dsply 'The file has not been opened';

(4)  on-Error ERR_DIVIDE_BY_ZERO:
      ERR_ARRAY_INDEX:
      ERR_DECIMAL_DATA;
      dsply 'You got a number wrong!';

(5)  on-Error *FILE;
      dsply 'You are doing something weird to a file';

(6)  on-Error *ALL;
      dsply 'Who knows what happened?';
      exSR *PSSR;
(1)  endMon;

*inLR = *on;

/include EXCEPTSRC,PSSRSTD

```

Note: You can create this program by running the following command:

```
CRTBNDRPG PGM(REDBOOK/ERROR06) SRCFILE(REDBOOK/EXCEPTSRC)
```

A number of changes have been made to the sample program, which are indicated by the numbers on the left side of Example 7-9 on page 260:

1. The code for which you want to trap exceptions errors is placed in a MONITOR group. If an exception/error is issued for any of the operations (between the MONITOR and the first ON-ERROR operation), control passes to the first ON-ERROR statement.
2. The Error Extender is removed from the first CHAIN operation because any error on the CHAIN is now caught by the MONITOR.
3. If the exception/error is “File not open”, a message is displayed and processing continues at the ENDMON operation.
4. If the exception/error is “Divide by zero”, “Array index error” or “Decimal data error”, a message is displayed and processing continues at the ENDMON operation.
5. If there is any other file exception/error (as opposed to program exception/error), a message is displayed and processing continues at the ENDMON operation.

6. If there is any other exception/error, a message is displayed and the *PSSR subroutine is run.

A MONITOR group also applies to code that is run in called subroutines. The MONITOR group in ERROR07 (Example 7-10) works in the exact same way as the MONITOR group in ERROR06.

Example 7-10 Source code for program ERROR07

```

dcl-F product1 disk(*ext) usage(*update: *delete) keyed usr0pn;

dcl-Pr fatalProgram extPgm('FATALPGM');
end-Pr;

dcl-S PSSRDone ind;
dcl-S dummyCode like(prodCd);
dcl-S a          int(10) inz(20);
dcl-S b          int(10);
dcl-S c          int(10);

/include EXCEPTSRC,ERRSTATUS

monitor;
c = a/b;

(1)      exSR subMonitor;

on-Error ERR_NOT_OPEN;
dsply 'The file has not been opened';

on-Error ERR_DIVIDE_BY_ZERO:
ERR_ARRAY_INDEX:
ERR_DECIMAL_DATA;
dsply 'You got a number wrong!';

on-Error *FILE;
dsply 'You are doing something weird to a file';

on-Error *ALL;
dsply 'Who knows what happened?';
exSR *PSSR;
endMon;

*inLR = *On;

(2)      begSR subMonitor;
chain dummyCode product1;

chain dummyCode product1;
open Product1;
endSR;

/include EXCEPTSRC,PSSRSTD

```

Note: You can create this program by running the following command:

```
CRTBNDRPG PGM(REDBOOK/ERROR07) SRCFILE(REDBOOK/EXCEPTSRC)
```

The file processing is placed in the subroutine SubMonitor (noted by the number 2 in Example 7-10 on page 262). Although the code of the subroutine is not physically in the MONITOR group, the fact that the EXSR is in the MONITOR group (noted by the number 1 in Example 7-10 on page 262) means that all of the code in the subroutine is monitored. This also applies to any subroutines that were run from the called subroutine, as shown in Example 7-11.

Example 7-11 Source code for subroutine for MONITOR group

```
dcl-F product1 disk(*ext) usage(*update: *delete) keyed usr0pn;

dcl-Pr fatalProgram extPgm('FATALPGM');
end-Pr;

dcl-S PSSRDone ind;
dcl-S dummyCode like(prodCd);
dcl-S a          int(10) inz(20);
dcl-S b          int(10);
dcl-S c          int(10);

/include EXCEPTSRC,ERRSTATUS

monitor;
c = a/b;

exSR subMonitor1;

on-Error ERR_NOT_OPEN;
  dsply 'The file has not been opened';

on-Error ERR_DIVIDE_BY_ZERO:
  ERR_ARRAY_INDEX:
  ERR_DECIMAL_DATA;
  dsply 'You got a number wrong!';

on-Error *FILE;
  dsply 'You are doing something weird to a file';

on-Error *ALL;
  dsply 'Who knows what happened?';
  exSR *PSSR;
endMon;

*inLR = *on;

begSR subMonitor1;
  c = a/b;
  exSR subMonitor2;
endSR;

begSR subMonitor2;
```

```

chain dummyCode product1;
chain dummyCode product1;
open product1;
endSR;
/include EXCEPTSRC,PSSRSTD

```

Note: You can create this program by running the following command:

```
CRTBNDRPG PGM(REDBOOK/ERROR08) SRCFILE(REDBOOK/EXCEPTSRC)
```

Although a MONITOR group may trap exception/errors from a subroutine that is run from within the group, the same cannot be said for a subprocedure that is called from within the group. A call to a subprocedure is akin to a call to a program, that is, it results in a new entry in the call stack. If the code in a subprocedure fails, you can trap the error on the call to the subprocedure by checking for a status of 00202 “Called program or procedure failed”. You can see how exception/error handling works with subprocedures in 7.5, “Subprocedures and exception/errors” on page 266.

7.4.4 Information data structures

If you need more information about the status of a program or any of the files in the program, you can use special data structures. The Program Status Data Structure (PSDS) provides information about the program’s status and File Information Data Structures (INFDS) provide information about the status of files. The information in these data structures is maintained by the RPG runtime routines as the program is running.

The PSDS is identified by an SDS definition (number 8 in Example 7-12) and there can only be one per program. An INFDS (number 1 in Example 7-12) is associated with a specific file by using the **INFDS** keyword on the F Spec. The INFDS must be unique for a file, that is, you cannot share a file information data structure between two files. These data structures contain an immense amount of information about the program and files and not just information relating to exception/errors.

For a complete list of the contents of the data structures, see the RPG IV Concepts → File and Program Exception/Errors section in the IBM i Version 7.2 Programming IBM Rational Development Studio for i ILE RPG Reference guide:

http://www.ibm.com/support/knowledgecenter/en/ssw_ibm_i_72/rzasd/sc092508.pdf?view=kc

Example 7-12 shows an example of a program status and a file information data structure.

Example 7-12 Program status and file information data structure

```

dcl-F screens workStn(*ext) usage(*input: *output)
(1)      infDS(workStnInfDS) infSR(*PSSR);
dcl-F database disk(*ext) usage(*update: *delete: *output)
(1)      keyed infDS(DBFfileInfDS) infSR(*PSSR);

(2)      dcl-Ds workStnInfDS noOpt qualified ;
(3)          status *status;
(4)          msgId    char(7) pos(40);
(5)          cursorRow int(3) pos(370);
(5)          CursorCol int(3) pos(371);
(5)          min_RRN   int(5) pos(378);
end-Ds;

```

```

(2)  dcl-Ds DBFileInfoDS noOpt qualified;
(3)    status *status;
(6)    lockedRecords int(5) pos(377);
(6)    RRN           int(5) pos(397);
      end-Ds;

(7)  dcl-Ds programStatus NoOpt PSDS qualified;
(8)    procedureName *proc;
(4)    msgId        char(7) pos(46);
      end-Ds;

```

Note the following points in Example 7-12 on page 264 as indicated by the numbers on the left side of the example:

1. The **INFDS** keyword identifies the data structure to be used as an INFDS.
2. The information data structure should normally be defined by using the **NOOPT** keyword, which ensures that, when a program is optimized, the latest values are applied in the data structure. The optimizer may keep some values in registers and restore them to storage only at predefined points during normal program execution. The use of **NOOPT** ensures that the values are always current. Although the use of **NOOPT** probably makes no difference for programs or modules that are compiled by using default optimization, it is advisable to always specify it for file or program information data structures in case the program or module is ever re-created for a higher optimization level. For more information about optimization, see the **OPTIMIZE** parameter on the **CRTBNDRPG** and **CRTRPGMOD** CL commands. The INFDS is qualified to ensure that there are not any name conflicts with sub-fields in the data structure.
3. Some of the definitions in the data structures are no longer required because they were replaced by BIFs, for example, the status code for a file or program can be determined using the **%STATUS** BIF.
4. The MSGID fields identify the relevant CPF or MCH error message that is received by the program. Note the use of the **OVERLAY** keyword to avoid the need to use From/To positioning on the D-spec.
5. Parts of the file information data structures are different depending on the type of the file. For a display file, the CURSORROW field identifies the row on the screen at which the cursor was placed when the screen was input. The CURSORCOL field identifies the column on the screen at which the cursor was placed when the screen was input. The MIN_RRN field identifies the RRN of the subfile record at the top of the screen when it was input (which is a lot more dependable than using the **SFLCSRRRN** keyword in DDS).
6. For a database file, the LOCKRECORDS field identifies the number of records currently locked. The RRN field identifies the relative record number of the current record.
7. SDS identifies the data structure as being a program status data structure.
8. PROCEDURENAME identifies the name of the program. Keywords may be used in place of the length and overlay positions for certain information, such as ***PROC** for the procedure name.

Program and file information data structures are another item that lend themselves to standard definitions in a copy member. Your first inclination might be to include the complete definition of the data structures, but this is not a preferred practice. The standard definitions should contain only the minimum information that you require; certain information in the data structures take time to obtain, so you should not define it unless you might use it.

7.4.5 ILE condition handlers

Section “Call stack and error handling” on page 121 describes in detail ILE condition handles. 7.6, “ILE CEE APIs” on page 287 does provide a simple example of an ILE condition handler being used along with some of the other ILE APIs.

7.5 Subprocedures and exception/errors

Perhaps you managed to get exception/error handling working in all of your programs and you start to experiment with subprocedures. Suddenly, exception/error handling does not seem to work in the same way.

You must remember that subprocedures are a feature of ILE. They are a form of “sub-program” and they result in their own entry in the call stack.

The other main consideration is that there is a major difference in the way exception/error handling works in the ILE runtime environment as opposed to OPM. In an ILE environment, exception/errors are “percolated” up the call stack.

Percolating messages mean that it is even easier to implement comprehensive exception/error handling in ILE than it is in OPM.

7.5.1 Percolation

An overview of percolation is provided in “Call stack and error handling” on page 121, here is a brief explanation from the point of view of handling exception/errors in an application.

In an OPM environment, a program that is down the call stack receives an exception/error. If the program does not have any exception/error handling, the RPG default handler issued a function check message.

In an ILE environment, a subprocedure or a program that is down the call stack receives an exception/error. If the subprocedure or program does not have any exception/error handling specified, the message is sent back up the call stack to the calling procedure. If the calling procedure does not have any exception/error handling defined, the message is again sent back up the call stack to the calling procedure and so on until the activation group control boundary is reached.

Messages are percolated to an activation group control boundary, which might be the program that originally initiated the AG. For more information, see the Control Boundaries section in the ILE Advanced Concepts manual:

https://www.ibm.com/support/knowledgecenter/ssw_ibm_i_72/ilec/ileac.htm

If none of the subprocedures or programs in the call stack has exception/error handling defined, the message is re-signaled as an exception and control returns to the subprocedure or program that originally received the message and an attempt is made to call the RPG default handler. But, subprocedures do not have a default RPG handler, which means that the call to the subprocedure causes an exception/error which, in turn, is percolated up the call stack. This process continues until a default handler can be called, which means that a function check may not be issued until an entry in the call stack is a mainline program (which has a default error handler).

Look at three examples of how exception/errors are percolated when none of the subprocedures or programs are using exception/error handlers (*PSSR, Error Extender, or Monitor Group).

Example 1

ERROR10 is another version of the sample program used throughout this chapter, and is shown in Example 7-13.

Example 7-13 Source code for example ERROR10

```
(1)    ctl-Opt dftActGrp(*no) actGrp('TEST1')
          option(*srcStmt: *noDebugIO);

          dcl-F product1 disk(*ext) usage(*update: *delete) keyed usr0pn;

          dcl-Pr ProgramProc;
          end-Pr;
          dcl-Pr FileProc;
          end-Pr;

(2)    programProc();
          fileProc();
          *inLR = *on;

          dcl-Proc programProc;
              dcl-Pi programProc;
              end-Pi;

          dcl-S a int(10) inz(20);
          dcl-S b int(10);
          dcl-S c int(10);

(3)    c = a/b;
          end-Proc;

          dcl-Proc fileProc;
              dcl-Pi fileProc;
              end-Pi;

          dcl-S dummyCode like(prodCd);

              chain dummyCode product1;
              open product1;
          end-Proc;
```

Note: You can create this program by running the following command:

CRTBNDRPG PGM(REDBOOK/ERROR10) SRCFILE(REDBOOK/EXCEPTSRC)

Note the following points in Example 7-13 as indicated by the numbers on the left side of the example:

1. The program is an ILE program because it contains subprocedures; therefore, it cannot be created with **DFTACTGRP(*YES)**.

2. All exception/error handling is removed and the processing logic is placed in two subprocedures: ProgramProc and FileProc.
3. The divide by zero error was re-introduced.

When you run this program, you might expect it to fail with the “RNQ0102 Attempt to divide by zero” message, as noted by 3 in Example 7-13, but it does not. Instead, the program fails with a “RNQ0202 The call to PROGRAMPRO ended in error (C G D F)” message, noted by 2 in Example 7-13 on page 267.

You see the following information in the joblog:

```
Attempt made to divide by zero for fixed point operation.
Function check. MCH1211 unmonitored by ERROR10 at statement 0000002100,
instruction X'0000'.
The call to PROGRAMPRO ended in error (C G D F).
```

The “divide by zero” message was percolated up the call stack. Because there is no exception/error handler and the ProgramProc subprocedure does not have a default RPG exception/error handler, the call to ProgramProc ends in error. This again causes an exception/error, which results in a function check because the call was issued from the mainline.

Example 2

ERROR11 (Example 7-14) is similar to ERROR10. The only difference is that the subprocedures ProgramProc is called from the subprocedure Nest2, which is called from the subprocedure Nest1.

Example 7-14 Sample code for example ERROR11

```
ctl-Opt dftActGrp(*no) actGrp('TEST2')
option(*srcStmt: *noDebugIO);

dcl-F product1 disk(*ext) usage(*update: *delete) keyed usr0pn;

(1)   nest1();
      fileProc();
      *inLR = *on;

      dcl-Proc nest1;
          dcl-Pi nest1 end-Pi;

          nest2();
      end-Proc;

      dcl-Proc nest2;
          dcl-Pi nest2 end-Pi;
          programProc();
      end-Proc;

      dcl-Proc programProc;
          dcl-Pi programProc end-Pi;

          dcl-S a int(10) inz(20);
          dcl-S b int(10);
          dcl-S c int(10);
```

```

        c = a/b;
end-Proc;

dcl-Proc fileProc;
dcl-Pi fileProc end-Pi;

dcl-S dummyCode like(prodCd);

chain dummyCode product1;
open product1;
end-Proc;

```

Note: You can create this program by running the following command:

```
CRTBNDRPG PGM(REDBOOK/ERROR11) SRCFILE(REDBOOK/EXCEPTSRC)
```

This time the program fails with a “RNQ0202 The call to NEST1 ended in error (C G D F)” message, as noted by 1 in Example 7-14. You see the following information in the joblog:

```

Attempt made to divide by zero for fixed point operation.
Function check. MCH1211 unmonitored by ERROR11 at statement 0000003700,
instruction X'0000'.
The call to NEST1 ended in error (C G D F).

```

The process here is the same as it was in ERROR10. All that was introduced is more levels in the call stack. Even though there are more levels involved, the end result is the same and the function check is issued for the mainline call to NEST1.

Example 3

The third example splits the sample program into two programs. ERROR15 (Example 7-15) starts an activation group and makes a dynamic call.

Example 7-15 Source code for example ERROR15

```

(1)   ctl-Opt dftActGrp(*no) actGrp('TEST2')
      option(*srcStmt: *noDebugIO);

(2)   dcl-Pr nest1 extPgm('ERROR16');
      end-Pr;

(3)   nest1();
      *inLR = *on;

```

Note: You can create this program by running the following command:

```
CRTBNDRPG PGM(REDBOOK/ERROR15) SRCFILE(REDBOOK/EXCEPTSRC)
```

Note the following points in Example 7-15 as indicated by the numbers on the left side of the example:

1. The program starts an activation group named TEST2.
2. The Nest1 prototype identifies a dynamic call to the program ERROR16. It is important to note that this is a dynamic call to another program as opposed to a bound call to a subprocedure.

3. The program issues a call to Nest1. If any operation further down the call stack should fail and is not handled, then the exception/error message arrives at this program.

Example 7-16 shows the ERROR16 program that is called from ERROR15.

Example 7-16 Source code for example ERROR16

```
(1)    ctl-Opt dftActGrp(*no) actGrp(*caller)
          option(*srcStmt: *noDebugIO);

          dcl-F product1 disk(*ext) usage(*update: *delete) keyed usr0pn;

          dcl-Pr nest1 extPgm('ERROR16')
          end-Pr;

          dcl-Pi nest1 end-Pi;

(2)    nest2();
          fileProc();
          *inLR = *on;

          dcl-Proc nest2;
          dcl-Pi nest2 end-Pi;

          programProc();
          end-Proc;

          dcl-Proc programProc;
          dcl-Pi programProc end-Pi;

          dcl-S a int(10) inz(20);
          dcl-S b int(10);
          dcl-S c int(10);

          c = a/b;
          end-Proc;

          dcl-Proc fileProc;
          dcl-Pi fileProc end-Pi;

          dcl-S dummyCode like(prodCd);

          chain dummyCode product1;
          open product1;
          end-Proc;
```

Note: You can create this program by running the following command:

CRTBNDRPG PGM(REDBOOK/ERROR16) SRCFILE(REDBOOK/EXCEPTSRC)

Note the following points in Example 7-16 as indicated by the numbers on the left side of the example:

1. The program is created with an activation group of *CALLER, that is, this program runs in the same activation group (TEST3 in this example) as the procedure that called it.

2. The program follows the same broad logic as before. The mainline issues procedure calls to Nest2 and FileProc. Nest2 calls ProgramProc.

When the first program is called, the second program fails with a “RNQ0202 The call to NEST2 ended in error (C G D F)” message, as noted by 2 in Example 7-16. You see the following information in the joblog:

```
Attempt made to divide by zero for fixed point operation.
Function check. MCH1211 unmonitored by ERROR16 at statement 0000003300,
instruction X'0000'.
The call to NEST2 ended in error (C G D F).
```

The failure point is in the second program because there is a default RPG exception/error handler that handles the error in the mainline.

7.5.2 Trapping percolated errors

When you start to use subprocedures, the effects of percolation on exception/errors can be a little disconcerting because the apparent point of failure is higher up the call stack than the actual point where the exception/error occurred, especially when the subprocedure that failed is in another module or service program.

However, handling exception/errors in an ILE environment is much easier than in an OPM environment, with a couple of minor differences.

You want your default exception/error handler to provide an orderly means of aborting a process. Because of percolation, the only programs or procedures that require an exception/error handler are those that mark the control boundary of an AG.

In “Example 3” on page 269, simply adding the standard *PSSR subroutine to the mainline of the first program means that the *PSSR subroutine handles any unhandled exception/error that is received by any program or subprocedure further down the call stack. ERROR17, shown in Example 7-17, is the same as ERROR15, but with the standard *PSSR subroutine added.

Example 7-17 Source code for example ERROR17

```
ctl-Opt dftActGrp(*no) actGrp('TEST4')
option(*srcStmt: *noDebugIO);

dcl-Pr nest1 extPgm('ERROR16');
end-Pr;

dcl-S PSSRDone ind;

dcl-Pr fatalProgram extPgm('FATALPGM');
end-Pr;

nest1();
*inLR = *on;

/include EXCEPTSRC,PSSRSTD
```

Note: You can create this program by running the following command:

```
CRTBNDRPG PGM(REDBOOK/ERROR17) SRCFILE(REDBOOK/EXCEPTSRC)
```

The divide by zero error generates the following messages in the joblog:

Attempt made to divide by zero for fixed point operation.
RPG status 00202 caused procedure ERROR17 in program REDBOOK/ERROR17 to stop.

The file error generates the following messages in the joblog:

I/O operation was applied to closed file PRODUCT1.
RPG status 00202 caused procedure ERROR17 in program REDBOOK/ERROR17 to stop.

Here are the advantages of using this method:

- ▶ The programs or procedures that mark the control boundary for an activation group are the only programs or procedures that require traditional exception/error handling and the inclusion of a *PSSR subroutine.
- ▶ You do not need to specify the *INFSR keyword on F specs. Actually, you cannot compile a module that specifies an INFSR for a file that is referenced in a subprocedure; the compiler issues the following message:
RNF5416 The subprocedure calculation specification refers to a file that has the INFSR keyword specified

Here are the disadvantages of using this method:

- ▶ Because all programs and subprocedures now share a common *PSSR, you no longer get a dump listing of the subprocedure or program that failed. But, programs and procedures further down the call stack might still have their own exception/error handling, so in the case where you feel a dump listing might be required, code the appropriate routines where they are needed.
- ▶ You must be careful that an exception/error that you expect to percolate back to the starting program is not inadvertently trapped by a program or subprocedure further down the call stack. This is especially true with calls to subprocedures in a monitor group or CALL operations with an Error Extender. An example of inadvertently trapping exception/errors is shown in 7.9.5, "The problem with throw and catch" on page 308.
- ▶ You need to identify which program or subprocedure originally received the exception/error. For example, if a subprocedure 20 levels down in the call stack fails with a divide by zero message, how does the first entry in the call stack identify the original message, line number, program (or service program) and procedure that received the message?

7.5.3 Identifying a percolated message

The message details for any message in the joblog shows details of the program, module, procedure and line number that received the message. For example, the message details for the RNX1211 message show the following details:

```
To program . . . . . : ERROR16
To library . . . . . : REDBOOK
To module . . . . . : ERROR16
To procedure . . . . . : FILEPROC
To statement . . . . . : 4200
```

But when it comes to identifying exception/errors that caused a program or subprocedure to fail, you should not depend on messages in the joblog. The amount of detail that is shown in a joblog depends on the logging level for the job (for more information, see the help for the

LOG parameter on the Change Job (**CHGJOB**) command). For example, no exception/errors are shown in the joblog for a job with a logging level of **LOG(1 00 *SECLVL)**.

A joblog contains a filtered view of the messages that were sent to program/procedure message queues in the job, which means that you can write a subprocedure that retrieves the percolated message from the program/procedure message queue of the program that starts the activation group.

This section describes a slightly more complex exception/error reporting program to work with the CL FATALPGM written earlier. The RPGLE program FATALPGMA uses the Receive Program Message (**QMHRCPGM**) API to retrieve the percolated message and some of the Dynamic Screen Management APIs to capture an image of the current screen that is displayed. A report containing details of the error message and an image of the screen is produced.

The member STDMMSGINFO contains the standard prototype and data structure definitions that are shown in Example 7-18.

Example 7-18 Source code for example STDMMSGINFO

```
// Standard API Error data structure used with most APIs
(1) dcl-Ds APIError qualified;
    bytesProvided int(10) inz(%size(APIError));
    bytesAvail    int(10) inz(0);
    msgId        char(7);
    *n           char(1);
    msgData      char(240);
end-Ds;

//-----
// Message APIs
//-----

// Receieve Message from Program Message Queue
(2) dcl-Pr receiveMsg extPgm('QMHRCPGM');
(3)   msgInfo      char(3000) options(*varSize);
        msgInfoLen   int(10) const;
(3)   formatName   char(8) const;
        callStack     char(10) const;
        callStackCtr int(10) const;
        msgType      char(10) const;
        msgKey       char(4) const;
        waitTime     int(10) const;
        msgAction    char(10) const;
        errorForAPI like(APIError);
end-Pr;

// Send Message to Program Message Queue
dcl-Pr sndPgmMsg extPgm('QMHSNDPM');
    errorMsgId   char(7) const;
    msgFile      char(20) const;
    msgData      char(3000) const;
    msgDtaLen    int(10) const;
    msgType      char(10) const;
    callStack    char(19) const;
    callStackCtr int(10) const;
```

```

    msgKey      char(4);
    errorForAPI like(APIError);
end-Pr;

//-----
// ILE CEE APIs
//-----

// Register a Condition Handler
dcl-Pr registerHandler extProc('CEEHDLR');
    pConHandler pointer(*proc) const;
    commArea    pointer        const;
    feedback    char(12)       options(*omit);
end-Pr;

// Un-Register a Condition Handler
dcl-Pr unRegisterHandler extProc('CEEHDLU');
    pConHandler pointer(*proc) const;
    feedback    char(12)       options(*omit);
end-Pr;

// Normal End of AG
dcl-Pr endAG extProc('CEETREC');
    language_RC int(10) const options(*omit);
    user_RC     int(10) const options(*omit);
end-Pr;

// Register Call Stack Entry Termination User Exit Procedure
dcl-Pr registerCancelStack extProc('CEERTX');
    pCancelProc pointer(*proc) const;
    CommArea    pointer        const options(*omit);
    Feedback    char(12)       options(*omit);
end-Pr;

// Un-Register Call Stack Entry Termination User Exit Procedure
dcl-Pr unRegisterCancelStack extProc('CEEUTX');
    pCancelProc pointer(*proc) const;
    feedback    char(12)       options(*omit);
end-Pr;

// Register Activation Group Exit Procedure
dcl-Pr registerCancelAG extProc('CEE4RAGE');
    pCancelProc pointer(*proc) const;
    feedback    char(12)       options(*omit);
end-Pr;

//-----
(4) // Dynamic Screen Manager APIs
//-----
// Create Input Buffer
dcl-Pr createInputBuffer int(10) extProc('QsnCrtInpBuf');
    bufferSize   int(10) const;
    increment    int(10) const options(*omit);
    maximumSize int(10) const options(*omit);

```

```

        bufferHandle int(10)      options(*omit);
        error         like(APIError) options(*omit);
end-Pr;

// Read Screen
dcl-Pr ReadScreen int(10) extProc( 'QsnReadScr' );
    bytesRead      int(10)      options(*omit);
    bufferHandle   int(10) const options(*omit);
    cmdBufferhandle int(10) const options(*omit);
    environmentHandle int(10)   options(*omit);
    error          like(APIError) options(*omit);
end-Pr;

// Retrieve pointer to data in input buffer
dcl-Pr RetrieveDataPtr pointer extProc( 'QsnRtvDta' );
    bufferHandle int(10) const;
    datapointer  pointer      options(*omit);
    error         like(APIError) options(*omit);
end-Pr;

//-----
// Throw/Catch Procedures
//-----

// Throw a message up the call stack
dcl-Pr throw extProc('throw');
    msgId    char(7)      const;
    msgDataIn varchar(3000) const options(*omit:*noPass);
    msgFileIn char(20)     const options(*noPass);
end-Pr;

// Catch a thrown message
dcl-Pr catch extProc('CATCH');
    caughtMessage LikeDS(base_CaughtMessage);
end-Pr;

(5)  dcl-S DummyPtr pointer;
     // DS returned by QMHRCVPM for format RCVM0300
(6)  dcl-Ds RCVM0300 qualified template;
     byteReturned  int(10);
     byteAvail    int(10);
     msgSeverity  int(10);
(7)  msgId        char(7);
     msgType      char(2);
     msgKey       char(4);
     msgFileName  char(10);
     msgLibSpec   char(10);
     msgLibUsed   char(10);
     alertOption  char(9);
     CCSIDCnvIndText int(10);
     CCSIDCnvIndData int(10);
     CCSIDMsg    int(10);
     CCSIDReplace int(10);
(8)  lenReplace1 int(10);
     lenReplace2 int(10);

```

```

(8)      lenMsgReturn    int(10);
(8)      lenMsgAvail     int(10);
(8)      lenHelpReturn   int(10);
(8)      lenHelpAvail    int(10);
(8)      lenSenderReturn int(10);
(8)      lenSenderAvail  int(10);
(9)      msgData         char(5000);
end-Ds;

// Sender structure returned in RCVM0300
(10)     dcl-Ds RCVM0300SndRcvInfo qualified template;
          sendingJob           char(10);
          sendIngJobProfile    char(10);
          sendingJobNo          char(6);
          dateSent              char(7);
          timeSent              char(6);
          sendingType            char(1);
          receivingType          char(1);
          sendingPgm             char(12);
          sendingModule          char(10);
          sendingProcedure        char(256);
          *n                     char(1);
          noStateNosSending     int(10);
          stateNosSending       char(30);
(11)     receivingPgm        char(10);
(11)     receivingModule     char(10);
(11)     receivingProcedure  char(256);
          *n                     char(10);
          noStateNosReceiving   int(10);
          stateNosReceiving     char(30);
          *n                     char(2);
          longSendingPgmNameOffset int(10);
          longSendingPgmNameLength int(10);
          longSendingProcNameOffset int(10);
          longSendingProcNameLength int(10);
          longReceivingProcNameOffset int(10);
          longReceivingProcNameLength int(10);
          microSeconds           char(6);
          sendingUsrPrf          char(10);
          names                  char(4000);
end-Ds;

// Condition Token passed to/from ILE Handler
dcl-Ds base_ConditionToken qualified template;
          msgSev    int(5);
          msgNo     char(2);
          *n       char(1);
          msgPrefix char(3);
          msgKey    char(4);
end-Ds;

// Message format returned by Catch
dcl-Ds base_CaughtMessage qualified template;
          msgId    char(7);
          msgFile  char(20);

```

```

        msgFileName char(10) overLay(msgFile);
        msgLibName  char(10) overLay(msgFile: *next);
        msgText    char(132);
        msgdata    char(3000);
end-Ds;

```

Note the following points in Example 7-18 on page 273 as indicated by the numbers on the left side of the example:

1. The standard API error structure is described in detail in Chapter 10, “Modern RPG comparison as viewed from a young developer” on page 365.
2. ReceiveMsg is the prototype for the Receive Program Message API (QMHRCPVM).
3. Message information is returned in the **MsgInfo** parameter in the format that is specified in the **FormatName** parameter. The required format of **MsgInfo** is defined by the RCVM0300 data structure (see notes 6 - 9).
4. Prototypes are defined for the Dynamic Screen Management (DSM) APIs that are used to retrieve an image of the current screen: Create Input Buffer (**QsnCrtInpBuf**), Read Screen (**QsnReadScr**), and Retrieve Data Pointer (**QsnRtvDta**).
5. Definitions of the structures that are required by the QMHRCPVM API are based on a pointer that is never set. This means that although included in a program, the data structures never occupy any memory. Programs that require these formats can define corresponding data structures by using the **LIKEDS** keyword.
6. The RCVM0300 data structure defines the format of the data returned by the QMHRCPVM API for the RCVM0300 format name.
7. The **MsgId** field is the message ID of the received message.
8. The sum of **LenReplace1**, **LenMsgReturn**, and **LenHelpReturn** provide the offset to the position of the sender information in the **MsgData** parameter.
9. The **MsgData** field contains details of message data (the number of characters that is defined by **LenReplace1**) followed by the first level message text (the number of characters that is defined by **LenMsgReturn**) followed by the second level message text (the number of characters that is defined by **LenHelpReturn**) followed by the sender/receiver data (the number of characters that is defined by **LenSenderReturn**). The sender/receiver data is further defined as a structure (next item).
10. The RCVM0300SenderReceiverInfo data structure defines the structure of the sender/receiver information returned in a portion of the **MsgData** field in the RCVM0300 data structure.
11. These fields identify the program (ReceivingPgm), module (ReceivingModule), procedure (ReceivingProcedure) and statement number (StateNosReceiving) that originally received the exception/error message. Interestingly enough, the sender/receiver information does not identify the library for the program.

FatalError (program FATALPGMA) produces a report detailing the exception/error details. Example 7-19 shows the DDS for the ERRORLST print file.

Example 7-19 DDS source for ERRORLST print file

```

A* ERRORLST - Exception Error Report
A          R HEAD
A
A          SKIPB(03)
A          4'Page: '
A          +0PAGNBR EDTCDE(Z)
A          33'Exception Error Report'

```

```

A           SPACEA(2)
A           4'Date: '
A           +0DATE
A           EDTCDE(Y)
A           21'Time: '
A           +0TIME
A           37'Job: '
A           JOBNO      6  0  +0
A           USER        10 0  +1
A           NAME        10 0  +1
A           SPACEA(2)
A           4'An unexpected exception error was -
A           detected in a program'
A           SPACEA(1)
A           4'The message details are:'
A           R DETAIL
A           SPACEB(2)
A           4'Program: '
A           PROGRAM     10 0  +0SPACEA(1)
A           4'Module   '
A           MODULE      10 0  +0SPACEA(1)
A           4'Procedure: '
A           PROCEDURE    60 0  +0SPACEA(1)
A           4'Statement: '
A           STATEMENT    10 0  +0SPACEA(2)
A           MSGTEXT      80 0  1SPACEA(2)
A           HLPTEXT01    80 0  1SPACEA(1)
A           HLPTEXT02    80 0  1SPACEA(1)
A           HLPTEXT03    80 0  1SPACEA(1)
A           HLPTEXT04    80 0  1SPACEA(1)
A           HLPTEXT05    80 0  1SPACEA(1)
A           HLPTEXT06    80 0  1SPACEA(1)
A           HLPTEXT07    80 0  1SPACEA(1)
A           HLPTEXT08    80 0  1SPACEA(1)
A           HLPTEXT09    80 0  1SPACEA(1)
A           HLPTEXT10    80 0  1SPACEA(1)
A           R SCREEN80
A           SPACEB(2)
A           1'-----'
A           +10'-----'
A           +10'-----'
A           +10'-----'
A           SPACEA(1)
A           ROW8001      80  1SPACEA(1)
A           ROW8002      80  1SPACEA(1)
A           ROW8003      80  1SPACEA(1)
A           ROW8004      80  1SPACEA(1)
A           ROW8005      80  1SPACEA(1)
A           ROW8006      80  1SPACEA(1)
A           ROW8007      80  1SPACEA(1)
A           ROW8008      80  1SPACEA(1)
A           ROW8009      80  1SPACEA(1)
A           ROW8010      80  1SPACEA(1)
A           ROW8011      80  1SPACEA(1)
A           ROW8012      80  1SPACEA(1)
A           ROW8013      80  1SPACEA(1)

```

A	ROW8014	80	1SPACEA(1)
A	ROW8015	80	1SPACEA(1)
A	ROW8016	80	1SPACEA(1)
A	ROW8017	80	1SPACEA(1)
A	ROW8018	80	1SPACEA(1)
A	ROW8019	80	1SPACEA(1)
A	ROW8020	80	1SPACEA(1)
A	ROW8021	80	1SPACEA(1)
A	ROW8022	80	1SPACEA(1)
A	ROW8023	80	1SPACEA(1)
A	ROW8024	80	1SPACEA(1)
A			1'-----'
A			+10'-----'
A			+10'-----'
A			+10'-----'
A	R SCREEN132		SPACEB(2)
A			1'-----'
A			+10'-----'
A			SPACEA(1)
A	ROW13201	132	1SPACEA(1)
A	ROW13202	132	1SPACEA(1)
A	ROW13203	132	1SPACEA(1)
A	ROW13204	132	1SPACEA(1)
A	ROW13205	132	1SPACEA(1)
A	ROW13206	132	1SPACEA(1)
A	ROW13207	132	1SPACEA(1)
A	ROW13208	132	1SPACEA(1)
A	ROW13209	132	1SPACEA(1)
A	ROW13210	132	1SPACEA(1)
A	ROW13211	132	1SPACEA(1)
A	ROW13212	132	1SPACEA(1)
A	ROW13213	132	1SPACEA(1)
A	ROW13214	132	1SPACEA(1)
A	ROW13215	132	1SPACEA(1)
A	ROW13216	132	1SPACEA(1)
A	ROW13217	132	1SPACEA(1)
A	ROW13218	132	1SPACEA(1)
A	ROW13219	132	1SPACEA(1)
A	ROW13220	132	1SPACEA(1)
A	ROW13221	132	1SPACEA(1)
A	ROW13222	132	1SPACEA(1)
A	ROW13223	132	1SPACEA(1)
A	ROW13224	132	1SPACEA(1)
A	ROW13225	132	1SPACEA(1)
A	ROW13226	132	1SPACEA(1)
A	ROW13227	132	1SPACEA(1)
A			1'-----'
A			+10'-----'
A			+10'-----'

```

A          +10'-----'
A          +10'-----'
A          +10'-----'
A          +10'-----'
A          +10'-----'
A          +10'-----'
A          SPACEA(1)
A          R FOOTER
A          SPACEB(3)
A          4'*** Error details complete ***

```

Note: You can create this print file by running the following command:

```
CRTPRTF FILE(REDBOOK/ERRORLST) SRCFILE(REDBOOK/EXCEPTSRC)
```

The exception/error report simply lists the message details along with details of the program/procedure that received the message and, if the program is running interactively, a copy of the screen.

Example 7-20 shows the H, F, and D specs of the FATALPGMA program.

Example 7-20 The H, F, and D specs of the FATALPGMA program

```

(1)   ctl-Opt dftActGrp(*no) actGrp(*caller)
      option(*srcStmt: *noDebugIO);

(2)   dcl-F errorLst printer(*ext) usage(*output)
      extFile('REDBOOK/ERRORLST') usr0pn OF1Ind(overFlow);

(3)   /include EXCEPTSRC,STDMSGINFO

      dcl-Pr fatalError extPgm('REDBOOK/FATALPGMA');
      end-Pr;
(4)   dcl-Pr fatalProgram extPgm('REDBOOK/FATALPGM');
      end-Pr;

      dcl-Pi fatalError;
      end-Pi;

(5)   dcl-Ds hlpText;
      hlpText01;
      hlpText02;
      hlpText03;
      hlpText04;
      hlpText05;
      hlpText06;
      hlpText07;
      hlpText08;
      hlpText09;
      hlpText10;
      end-Ds;

      dcl-Ds screenIn;
      screenInData Char(3564);
      end-Ds;

      dcl-S p_screenIn pointer inz(%addr(screenIn));

```

```
dcl-Ds d_screen80 based(p_screenIn);
row8001;
row8002;
row8003;
row8004;
row8005;
row8006;
row8007;
row8008;
row8009;
row8010;
row8011;
row8012;
row8013;
row8014;
row8015;
row8016;
row8017;
row8018;
row8019;
row8020;
row8021;
row8022;
row8023;
row8024;
end-Ds;
dcl-Ds d_screen132 based(p_screenIn);
row13201;
row13202;
row13203;
row13204;
row13205;
row13206;
row13207;
row13208;
row13209;
row13210;
row13211;
row13212;
row13213;
row13214;
row13215;
row13216;
row13217;
row13218;
row13219;
row13220;
row13221;
row13222;
row13223;
row13224;
row13225;
row13226;
row13227;
end-Ds;
```

```

(6)    dcl-Ds programStatus psds;
        NAME pos(244);
        USER pos(254);
        JOBNO pos(264);
        end-Ds;

(7)    dcl-Ds msgBack likeDs(RCVM0300) inz;

        dcl-S infoPtr pointer;
(8)    dcl-Ds msgInfo likeDs(RCVM0300SndRcvInfo) based(infoPtr);

        dcl-S i           int(10);
        dcl-S setMsgKey   char(4) inz(*ALLx'00');
        dcl-S bufferHandle int(10);
        dcl-S bytesReturned int(10);
        dcl-S dataPtr     pointer;

        dcl-Ds catchScreen likeDS(screenIn) based(dataPtr);

```

Note the following points in Example 7-20 as indicated by the numbers on the left side of the example:

1. The H spec indicates that the program should run in the activation group of the caller.
2. The F spec for the ERRORLST print file uses the EXTFILE keyword to ensure that the print file can be found (in this case, it is better not to depend on the library list to locate the print file).
3. The STDMMSGINFO member containing prototypes and standard definitions is included.
4. The prototype for the original FATALPGM is included. FATALPGM is called after the error report is produced.
5. Data structures are used to map the print fields (from ERRORLST) to fields that are retrieved from APIs. The HlpText data structure is used to re-define the retrieved second level message text for printing. The ScreenIn data structure is used to re-define the retrieved screen image (either 24 by 80 or 27 by 132). Lengths do not need to be defined for the sub-fields because they are externally defined on the ERRORLST print file.
6. A program status data structure is used to identify the job.
7. MsgBack is the data structure that receives the exception/error message. The LIKEDS keyword is used to define it like the standard RCVM0300 data structure in STDMGINFO.
8. MsgInfo is the data structure that contains the sender/receiver information from the message data field in the MsgBack data structure. The LIKEDS keyword is used to define it like the standard RCVM0300SndRcvInfo data structure in STDMGINFO.

The program starts by retrieving and reporting the error, as shown in Example 7-21.

Example 7-21 Source code for FatalError subprocedure

```

(1)    open errorLst;
        write head;

(2)    receiveMsg( msgBack
                  : %size(msgBack)
                  : 'RCVM0300'
                  : '*'

```

```

        : 2
        : '*PRV'
        : setMsgKey
        : 0
        : '*SAME'
        : APIError);

(3)    If (msgBack.byteAvail > 0);

(4)    msgText = %subSt(msgBack.msgData:
                      msgBack.lenReplace1 + 1:
                      msgBack.lenMsgReturn);
    hlpText = %subSt(msgBack.msgData:
                      msgBack.lenReplace1 +
                      msgBack.lenMsgReturn + 1:
                      msgBack.lenHelpReturn);

(5)    infoPtr = %addr(msgBack.msgData)
          + msgBack.lenReplace1
          + msgBack.lenMsgReturn
          + msgBack.lenHelpReturn;
    program = msgInfo.receivingPgm;
    module = msgInfo.receivingModule;
    procedure = msgInfo.receivingProcedure;
    statement = msgInfo.stateNosReceiving;

(6)    write detail;
    if overFlow;
      write head;
      overFlow = *Off;
    endIf;

endIf;

```

Note the following points in Example 7-21 as indicated by the numbers on the left side of the example:

1. The FatalError subprocedure starts by opening the ERRORLST print file and printing the headings, which identifies the failing job.
2. The QMHRCVPM API is called to retrieve the last message in the program/procedure message queue of the calling program/procedure. The call stack counter is 2 because the calling procedure/program is two entries back in the call stack; a dynamic call is made to this program so it has a PEP in the call stack. The message action is set to *SAME to ensure that the message remains in the job log.
3. The ByteAvail field in the Msgback data structure identifies whether a message was retrieved from the message queue.
4. The first level and second level message text are retrieved from the MsgData field in the MsgBack data structure by using the returned lengths to determine the starting positions and lengths to retrieve.
5. The returned lengths are again used to set the basing pointer for the MsgInfo data structure so it is mapped on to the relevant part of the MsgData field in the MsgBack data structure.
6. The message details are printed.

The program continues by capturing the current screen image (if it is an interactive job), as shown in Example 7-22.

Example 7-22 Capturing the current screen image

```
(1)    bufferHandle = createInputBuffer( 27 * 132
                                         : *omit
                                         : *omit
                                         : *omit
                                         : APIError );

(2)    If (APIError.BytesAvail = 0);
(3)        bytesReturned = readScreen( *omit
                                         : bufferHandle
                                         : *omit
                                         : *omit
                                         : *omit );

(4)    dataPtr = retrieveDataPtr( bufferHandle
                                         : *omit
                                         : *omit );

(5)    screenIn = %subSt(catchScreen: 1: bytesReturned);
(6)    for i = 1 to bytesReturned;
        if (%subSt(screenIn:i:1) > x'19') and
           (%subSt(screenIn:i:1) < x'40');
           %subSt(screenIn:i:1) = *blank;
        endIf;
    endFor;
(7)    if (bytesReturned = 1920);
        write screen80;
    else;
        write screen132;
    endIf;
endIf;
```

Note the following points in Example 7-22 as indicated by the numbers on the left side of the example:

1. The Create Input Buffer API is called to get a buffer handle for the screen.
2. Call the remainder of the DSM APIs only if a buffer handle was successfully created.
3. Read the screen and determine the number of bytes returned, which will be 1920 for a 24 by 80 screen or 3564 for a 27 by 132 screen. This API captures an image of all characters on the screen.
4. Get a pointer to the buffer for the screen just read. The CatchScreen data structure is based on this pointer.
5. Copy the screen buffer (CatchScreen) to the ScreenIn data structure so that the print fields are populated.
6. In this example, any attribute bytes in the captured screen image are replaced with blanks. You can replace any attribute byte with the identifying character you deem appropriate, for example, to mark the beginning of entry fields.
7. Print either the 24 by 80 or 27 by 132 image of the screen, depending on the number of characters that are retrieved.

The program finishes by printing the footer, closing the print file, and calling the FatalProgram to notify the error and provide a meaningful message to the user, as shown in Example 7-23.

Example 7-23 Source for the final part

```
write footer;
    close errorLst;

    fatalProgram();
    *inLR = *on;
```

Note: You can create this program by running the following command:

```
CRTBNDRPG PGM(REDBOOK/FATALPGMA) SRCFILE(REDBOOK/EXCEPTSRC)
```

Using the FatalError program means that the *PSSR subroutine must change. Instead of calling the FatalPgm program, the *PSSR subroutine will call the FatalError program. ERROR18 (Example 7-24) is an altered version of ERROR17 with a new *PSSR coded.

Example 7-24 Source code for program ERROR18

```
ctl-Opt dftActGrp(*no) actGrp('TEST5')
option(*srcStmt: *noDebugIO);

dcl-Pr nest1 extPgm('ERROR16');
end-Pr;

(1)   dcl-Pr fatalError extPgm('REDBOOK/FATALPGMA');
end-Pr;

nest1();
*InLR = *On;

(2)   begSR *PSSR;
monitor;
    fatalError();
on-Error;
    dsply 'Agggghhhh!';
endMon;
endSR '*CANCL';
```

Note: You can create this program by running the following command:

```
CRTBNDRPG PGM(REDBOOK/ERROR18) SRCFILE(REDBOOK/EXCEPTSRC)
```

Note the following points in Example 7-24 as indicated by the numbers on the left side of the example:

1. The FatalError prototype identifies a dynamic call to the FATALPGMA program. The program name is qualified to remove any dependency on a library list.
2. The *PSSR subroutine is a much simpler version of the previous one in that all it does is call FatalError. All code in the *PSSR subroutine is placed in a MONITOR group to ensure there are not un-requested repeated calls to *PSSR.

Any unhandled exception/error that is detected in any program or procedure in the call stack results in the *PSSR routine above being run and the subsequent call to FatalError.

Figure 7-1 shows an example of the resulting error exception report that is produced when the file error is detected in the FileProc procedure in ERROR16.

```
Page: 1                               Exception/Error Report
Date: 29/11/06   Time: 11:34:41  Job: 056418 TUOHYP      COMCONPTA
An unexpected exception/error was detected in a program
The message details are:
Program:    ERROR16
Module:     ERROR16
Procedure:  FILEPROC
Statement:  4200
I/O operation was applied to closed file PRODUCT1.
Cause . . . . : RPG procedure FILEPROC in program REDBOOK/ERROR16 attempted
Operation CHAIN on file PRODUCT1 while the file was closed. Recovery . . . :
Contact the person responsible for program maintenance to determine the cause
of
the problem.
-----
MAIN           OS/400 Main Menu          System: S655D66B
Select one of the following:
1. User tasks
2. Office tasks
3. General system tasks
4. Files, libraries, and folders
5. Programming
6. Communications
7. Define or change the system
8. Problem handling
9. Display a menu
10. Information Assistant options
11. iSeries Access tasks
90. Sign off
Selection or command
====> call error18

F3=Exit   F4=Prompt   F9=Retrieve   F12=Cancel   F13=Information Assistant
F23=Set initial menu

-----
*** Error details complete ***
```

Figure 7-1 Resulting error exception report produced in FileProc procedure in ERROR16

You have seen how you can extend the functions of the program that is called when an unexpected exception/error occurs. This program can be extended even further to incorporate the automatic generation of a log number, recording the incident to a database file, communicating with the operator, or whatever else you deem appropriate.

7.6 ILE CEE APIs

The ILE Common Execution Environment (CEE) APIs can be useful when used with exception/error handling, specifically the condition management APIs and the activation group and control flow APIs.

This section covers the following topics:

- ▶ 7.6.1, “Condition management APIs” on page 287
- ▶ 7.6.2, “Activation group and control flow APIs” on page 289
- ▶ 7.6.3, “Further information” on page 298

For more information about the ILE CEE APIs, see the *ILE CEE APIs* topic in the IBM i 7.2 Knowledge Center:

http://www.ibm.com/support/knowledgecenter/ssw_ibm_i_72/apis/ile1a1.htm

7.6.1 Condition management APIs

The use of the relevant ILE Condition Management APIs was covered in the “ILE condition handlers” on page 266. For the sake of completeness, this section describes a simple ILE condition handler to demonstrate how it works with exception/error handling.

When a condition handler is registered, it applies to the program or procedure in which it is registered and to all programs or procedures further down the call stack. Condition handlers may be “nested” in that another condition handler may be registered in another program or procedure in the call stack.

One of the most useful features of an ILE condition handler is that it can determine whether it handles the exception/error, whether processing should continue, or whether the exception/error should be percolated up the call stack.

The STDMMSGINFO copy member contains the prototypes for the Register a Condition Handler (CEEHDLR) and Un-Register a Condition Handler (CEEHDLU) APIs and the base definition of a condition token, as shown in Example 7-25.

Example 7-25 Source code for STDMMSGINFO copy member

```

//-----
// ILE CEE APIs
//-----

// Register a Condition Handler
dcl-Pr registerHandler extProc('CEEHDLR');
    pConHandler pointer(*proc) const;
    commArea    pointer        const;
    feedback    char(12)       options(*omit);
end-Pr;

// Un-Register a Condition Handler
dcl-Pr unRegisterHandler extProc('CEEHDLU');
    pConHandler pointer(*proc) const;
    feedback    char(12)       options(*omit);
end-Pr;

// Condition Token passed to/from ILE Handler

```

```
dcl-Ds base_ConditionToken qualified template;
  msgSev    int(5);
  msgNo     char(2);
  *n        char(1);
  msgPrefix char(3);
  msgKey    char(4);
end-Ds;
```

ERROR21 (Example 7-26) is an amended version of ERROR18 that includes the registering and un-registering of an ILE condition handler along with the coding of the condition handler itself.

Example 7-26 Source code for ERROR21

```
(1)   ctl-Opt dftActGrp(*no) actGrp('TEST6')
      option(*srcStmt: *noDebugIO);

      /include EXCEPTSRC,STDMSGINFO

      dcl-Pr nest1 extPgm('ERROR16');
      end-Pr;

      dcl-Pr fatalError extPgm('REDBOOK/FATALPGMA');
      end-Pr;

(2)   dcl-S pILEHandler pointer(*proc) inz(%pAddr('ILEHANDLER'));

(3)   registerHandler(pILEHandler : *null : *omit);

      nest1();

(4)   unRegisterHandler(pILEHandler : *omit);
      *inLR = *on;

      begSR *PSSR;
      monitor;
      fatalError();
      on-Error;
      dspla 'Agggghhhhh!';
      endMon;
      endSR '*CANCL';

      dcl-Proc ILEHandler export;
      dcl-Pi ILEHandler;
      tokenIn likeDS(Base_ConditionToken) const;
      pCommArea pointer const;
      action   int(10);
      tokenOut likeDS(base_ConditionToken);
      end-Pi;

      dcl-C RESUME 10;
      dcl-C PERCOLATE 20;

      if (tokenIn.msgPrefix = 'MCH' and
          tokenIn.msgNo = X'1211');
          action = RESUME;
```

```

        else;
(6)      action = PERCOLATE;
        endIf;
        Return;
end-Proc;

```

Note: You can create this program by running the following command:

```
CRTBNDRPG PGM(REDBOOK/ERROR21) SRCFILE(REDBOOK/EXCEPTSRC)
```

Note the following points in Example 7-26 on page 288 as indicated by the numbers on the left side of the example:

1. The inclusion of the bindable ILE CEE APIs does not require any special binding directory.
2. ILEHandler is the procedure pointer to the ILE condition handler that is registered in the program.
3. The ILEHandler subprocedure is registered as an ILE condition handler. It now applies to all further entries on the call stack unless superseded by another handler (Error Extender, MONITOR group, or another ILE condition handler). In this example, a communications area is not used, so the second parameter is null.
4. The ILE condition handler is unregistered at the end of the program.
5. The ILE condition handler indicates that all divide by zero errors should be ignored by resuming. It is not the usual course of action that is preferred because the resulting value is unpredictable, but it is applicable in the simple example you see here.
6. All other errors are percolated.

The end result of registering the ILEHandler condition handler is that the divide by zero exception/error in the ProgramProc subprocedure in ERROR16 is ignored, and any other exception/error results in the “normal” default exception/error process.

7.6.2 Activation group and control flow APIs

The ILE CEE activation group and control flow APIs provide features that might be useful in controlling the orderly termination of activation groups or the opportunity to run clean up or termination routines when a program or procedure is canceled (for example, close open files).

The following are some of the more commonly used activation group and control flow APIs:

- ▶ CEERTX: Register Call Stack Entry Termination user exit procedure
- ▶ CEEUTX: Un-Register Call Stack Entry Termination user exit procedure
- ▶ CEETREC: Normal End (of activation group)
- ▶ CEE4AGE: Register Activation Group exit procedure

The following sections shows an example that uses each of these APIs, but first here is a brief description of how they work.

CEETREC

CEETREC may be used to end an activation group “normally”. It may be called from any program or procedure within the activation group and the activation group is immediately ended much in the same way as the Reclaim Activation Group (**RCLACTGRP**) command can end an activation group. The main difference between using CEETREC and **RCLACTGRP** is that **RCLACTGRP** cannot reclaim the activation group in which the command is run.

In RPG, CEETREC provides the same functionality as the `exit()` function in C and the STOP RUN operation in COBOL.

Example 7-27 shows the prototype for CEETREC.

Example 7-27 CEETREC prototype

```
dcl-Pr endAG extProc('CEETREC');
  language_RC int(10) const options(*omit);
  user_RC    int(10) const options(*omit);
end-Pr;
```

The parameters to CEETREC are usually omitted. Both parameters are optional “return codes” which are input to CEETREC.

Although control is not returned to any of the entries in the call stack, any clean-up procedures registered for call stack entries are called before the activation group ends.

CEERTX and CEEUTX

You can use CEERTX to register a procedure that is called when the call stack entry for which it is registered is ended by anything other than a return to the caller. This situation might be caused by a call ending abnormally or a request to end the activation group by using CEETREC or **RCLACTGRP**, a request to end the job by using ENDJOB, or a user canceling the program or procedure by using option 2 from the system request menu.

Multiple exit procedures may be registered for a call stack entry. They are called in reverse order of registration (LIFO).

Example 7-28 shows the prototypes for CEERTX and CEEUTX.

Example 7-28 CEERTX and CEEUTX prototypes

```
// Register Call Stack Entry Termination User Exit Procedure
dcl-Pr registerCancelStack extProc('CEERTX');
  pCancelProc pointer(*proc) const;
  CommArea   pointer      const options(*omit);
  Feedback    char(12)      options(*omit);
end-Pr;

// Un-Register Call Stack Entry Termination User Exit Procedure
dcl-Pr unRegisterCancelStack extProc('CEEUTX');
  pCancelProc pointer(*proc) const;
  feedback     char(12)      options(*omit);
end-Pr;
```

Here are the parameters:

- ▶ The first parameter for both APIs is a procedure pointer to the procedure being registered or unregistered.
- ▶ The second parameter to CEERTX is a pointer to a communications area. Because you have no means of defining your own parameters with this API, the communications area provides a means of passing application specific data to the procedure.
- ▶ Both APIs have an ommissible feedback area.

The procedure being registered requires a prototype with the parameters that are shown in Example 7-29.

Example 7-29 Prototype parameters

```
dcl-Pr cancelStack;
    commArea pointer const;
end-Pr;
```

The single parameter is the pointer to the communications area that is specified when the procedure was registered.

CEE4AGE

You can use CEE4AGE to register a procedure that is called when an activation group ends. The procedure can perform any type of cleanup that you might deem necessary before the activation group is reclaimed, for example, write information to a log file, or clean up work files or user spaces.

A multiple exit point procedure may be registered. They are called in reverse order of registration (LIFO).

Example 7-30 shows the prototype for CEE4RAGE.

Example 7-30 CEE4RAGE prototype

```
// Register Activation Group Exit Procedure
dcl-Pr registerCancelAG extProc('CEE4RAGE');
    pCancelProc pointer(*proc) const;
    feedback     char(12)      options(*omit);
end-Pr;
```

The first parameter is a procedure pointer to the procedure being registered. The second parameter is an ommissible feedback area.

The procedure being registered is passed four parameters by the system, so it requires a prototype with the parameters that are shown in Example 7-31.

Example 7-31 Prototype parameters

```
dcl-Pr AGCancel;
    AGMark     int(10) const;
    reason     int(10) const;
    resultCode int(10);
    user_RC   int(10);
end-Pr;
```

Here are the parameters

- ▶ The first parameter is a marker that uniquely identifies the activation group within the job.
- ▶ You are primarily interested in the second parameter, which identifies why and how the activation group is ending. You see how this is interpreted in the upcoming example.
- ▶ The third parameter is a result code that is passed between activation group Exit procedures when multiple procedures are registered (a value of 0 is passed to the first exit procedure).

Table 7-3 shows the possible values for the result code.

Table 7-3 Result code values

Code	Meaning
0	Do not change the action.
10	Do not perform any pending error requests. This code is used if a previous exit procedure specified a result code of 20 and a subsequent procedure recovers from the error. The message CEE9901, indicating an application error, is not sent.
20	Send message CEE9901 to the caller of the control boundary after the remaining exit procedures are called.
21	Send message CEE9901 to the caller of the control boundary. The remaining exit procedures that are registered by the CEE4RAGE API are not called. This code is used if an unrecoverable error occurs in the exit procedure requesting this action.

- ▶ The fourth parameter is a user result code that is passed between activation group Exit procedures; set it as you see fit.

Example

This section shows an example of the Activation Group and Control Flow APIs in action. The member CANCEL01 contains global definitions, a mainline, and a subprocedure. Example 7-32 shows the global definitions and mainline.

Example 7-32 Global definitions and mainline

```
(1)    ctl-Opt dftActGrp(*no) actGrp('CANCEL')
          option(*srcStmt: *noDebugIO);

          /include EXCEPTSRC,STDMSGINFO

(2)    dcl-Pr subProc01 extPgm('CANCEL02');
          end-Pr;

          dcl-S x int(10);

(3)    x = x + 1;
          dsply x;

(4)    if x > 2;
          subProc01();
          endIf;
          return;

          begSR *inzSR;
(5)    registerCancelAG(%pAddr('AGCANCEL'): *omit);
          endSr;
```

Note the following points in Example 7-32 as indicated by the numbers on the left side of the example:

1. The program runs in a named activation group called CANCEL, which means that the activation group remains in the job when this program is exited.
2. The prototypes for the CEE APIs are placed in the copy member STDMSGINFO.
3. The program adds 1 to x and displays the value.

4. When x is greater than 2 (which happens when the program is three times in succession), a dynamic call is made to the program CANCEL02.
5. The subprocedure AGCancel is registered as an exit procedure for the activation group. This is performed in the *INZSR subroutine to ensure that the procedure is registered only once; if it were in the mainline, the procedure would be registered on every call and results in the procedure being called multiple times when the activation group ends.

The member CANCEL01 also contains the definition of the AGCancel subprocedure, as shown in Example 7-33. The subprocedure does not have to be placed in the same module that registers it, it can be placed in a module in a service program.

Example 7-33 Definition of the AGCancel subprocedure in CANCEL01

```

dcl-Proc AGCancel;
  dcl-Pi AGCancel;
    AGMark    int(10) const;
    reason    int(10) const;
    resultCode int(10);
    user_RC   int(10);
  end-Pi;

(1)   dcl-Ds *N;
      reason0 int(10);
      reason1 int(3) overlay(reason0);
      reason2 int(3) overlay(reason0: *next);
      reason3 int(3) overlay(reason0: *next);
      reason4 int(3) overlay(reason0: *next);
    end-Ds;

(2)   dcl-C END_NORMAL    128;
        dcl-C END_RCLACTGRP 32;
        dcl-C END_ENDJOB    16;
        dcl-C END_CEETREC    8;
        dcl-C END_FUNCGECK   4;
        dcl-C HEX_00          x'00';

      reason0 = reason;
      if (%bitAnd(reason3:END_NORMAL) = HEX_00);
        dspl 'AG ended normally';
      else;
        dspl 'AG ended abnormally';
      endIf;

      if (%bitAnd(reason3: END_RCLACTGRP) <> HEX_00);
        dspl 'AG ended by RCLACTGRP';
      elseif (%bitAnd(reason3: END_ENDJOB ) <> HEX_00);
        dspl 'AG ended by Job Ending';
      elseif (%bitAnd(reason3: END_CEETREC) <> HEX_00);
        dspl 'AG ended by exit request (CEETREC)';
      elseif (%bitAnd(reason3: END_FUNCGECK) <> HEX_00);
        dspl 'AG ended unhandled function check';
      else;
        dspl 'AG ended';
      endIf;
    end-Proc;

```

Note: You can create this program by running the following command:

```
CRTBNDRPG PGM(REDBOOK/CANCEL01) SRCFILE(REDBOOK/EXCEPTSRC)
```

Note the following points in Example 7-33 on page 293 as indicated by the numbers on the left side of the example:

- Even though the reason code is passed as an integer, you must decipher bit settings in the four bytes to determine why and how the activation group is ending. One way of achieving this task is to remap the four-byte integer to four separate bytes and use the %BITAND BIF to determine whether the relevant bits are set.

Table 7-4 is taken from the description of the CEE4RAGE API and provides the common reason codes for ending activation groups and call stack entries, with byte 3 (bits 16 - 23) being the most relevant.

Table 7-4 Common reason codes for ending activation groups and call stack entries

Bit	Meaning
Bits 0	Reserved.
Bits 1	Call stack entry is canceled because an exception message was sent.
Bits 2-15	Reserved.
Bit 16	0 - normal end 1 - abnormal end.
Bit 17	Activation Group is ending.
Bit 18	Initiated by the Reclaim Activation Group (RCLACTGRP) command.
Bit 19	Initiated as a result of the job ending.
Bit 20	Initiated by an exit verb, for example exit() in C, or the CEETREC API.
Bit 21	Initiated by an unhandled function check.
Bit 22	Call stack entry canceled because of an out-of-scope jump. For example, longjmp() in C.
Bits 23-31	Reserved.

- Named constants are used to identify the bit settings to be tested, for example, a value of 128 indicates that bit 0 of a byte is on or a value of 32 indicates that bit 2 of a byte is on.
- The %BITAND BIF is used to determine whether the relevant bits are set in the third byte of the reason code and to condition the relevant message being displayed.

You are now ready for the first test. Call the program CANCEL01 and the value of x is displayed:

```
DSPLY      1
```

Call CANCEL01 a second time and the value of x is again displayed:

```
DSPLY      2
```

Now issue the command **RCLACTGRP CANCEL** (do not call CANCEL01 for a third time) and the following is displayed:

```
DSPLY AG ended normally
DSPLY AG ended by RCLACTGRP
```

Reclaiming the activation group causes the AGCancel subprocedure to be called.

Now, add some additional complexity. The member CANCEL02 contains global definitions, a mainline, and a number of subprocedures. Example 7-34 shows the global definitions and mainline.

Example 7-34 Global definitions and mainline

```
(1) ctl-Opt dftActGrp(*no) actGrp(*caller)
           option(*srcStmt: *noDebugIO);

       /include EXCEPTSRC,STDMMSGINFO

(2) subProc02();
    return;

    begSR *inzSR;
(3)     registerCancelAG(%pAddr('AGCANCELNEXT'): *omit);
    endSr;
```

Note the following points in Example 7-34 as indicated by the numbers on the left side of the example:

1. The program runs in the activation group of the calling program or procedure. In this instance, this program runs in the CANCEL activation group.
2. The program simply calls the subprocedure SubProc02.
3. The subprocedure AGCancelNext is registered as an exit procedure for the activation group. This is the second exit procedure that is registered for the activation group (the first being in CANCEL01).

Example 7-35 shows the AGCancelNext subprocedure. It is a simplified version of the exit procedure in CANCEL01. All it does is display a message. This is a second exit procedure that is registered for the AG.

Example 7-35 AGCancelNext subprocedure

```
dcl-Proc AGCancelNext;
  dcl-Pi AGCancelNext;
    AGMark  int(10) const;
    reason   int(10) const;
    resultCode int(10);
    user_RC  int(10);
  end-Pi;

  dsply 'AGCancelNext in CANCEL02';
end-Proc;
```

Example 7-36 shows the CancelStack subroutine. It displays the message based on the pointer that is passed as the communication area. This program uses the same exit procedure for multiple call stack entries (because all it does is display a message), but you can have as many different exit procedures as required.

Example 7-36 CancelStack subroutine

```
dcl-Proc cancelStack;
  dcl-Pi cancelStack;
    commArea pointer const;
  end-Pi;
  dcl-S msg char(20) based(commArea);
    dspl msg;
  end-Proc;
```

Example 7-37 shows the SubProc02 subroutine, which is called from the mainline of CANCEL02. It registers CancelStack as an exit procedure with a pointer to the relevant message as the second parameter. It then issues a call to SubProc03.

Example 7-37 SubProc02 subroutine

```
dcl-Proc subProc02;
  dcl-Pi subProc02 end-Pi;
  dcl-S msg char(20) inz('Cancel subProc02');
  registerCancelStack(%pAddr(cancelStack)
    :%addr(msg)
    :*omit);
  // Your code here
  subProc03();
end-Proc;
```

Example 7-38 shows the SubProc03 subroutine, which is called from the SubProc02 subroutine.

Example 7-38 SubProc03 subroutine

```
dcl-Proc subProc03;
  dcl-Pi subProc03 end-Pi;
  dcl-S msg char(20) inz('Cancel subProc03');
(1)   registerCancelStack(%pAddr(cancelStack)
    :%addr(msg)
    :*omit);
  // There might be runnable code here
(2)   unRegisterCancelStack(%pAddr(cancelStack)
    :*omit);
  // Your code here
  monitor;
  subProc04();
  on-Error;
  endMon;
(3)   subProc05(0);
(4)   subProc05(1);
  end-Proc;
```

Note the following points in Example 7-38 on page 296 as indicated by the numbers on the left side of the example:

1. It registers CancelStack as an exit procedure with a pointer to the relevant message as the second parameter.
2. It un-registers the exit procedure after some code runs. The exit procedure would have been called if any of the operations between registering and un-registering caused the subprocedure to cancel.
3. It issues a call to SubProc04. The call is in a MONITOR group because the call is going to fail and you do not want the error to be percolated back up the call stack.
4. It issues a call to SubProc05 with a parameter value of 0.
5. It issues a call to SubProc05 with a parameter value of 1.

Example 7-39 shows the SubProc04 subprocedure, which is called from the SubProc03 subprocedure. It registers CancelStack as an exit procedure with a pointer to the relevant message as the second parameter. It then performs an invalid divide by zero that causes the call to the subprocedure to end in error, which in turn causes the exit procedure for the call stack entry to be called.

Example 7-39 SubProc04 subprocedure

```
dcl-Proc subProc04;
  dcl-Pi subProc04 end-Pi;
  dcl-S msg char(20) inz('Cancel SubProc04');
  dcl-S a int(10) inz(20);
  dcl-S b int(10);
  dcl-S c int(10);
  registerCancelStack(%pAddr(cancelStack)
    :%addr(msg)
    :*omit);
  // Your code here
  c = a/b;
end-Proc;
```

Example 7-40 shows the SubProc05 subprocedure, which is called twice from the SubProc03 subprocedure. It registers CancelStack twice as an exit procedure with a pointer to the relevant message as the second parameter. The subprocedure ends the activation group if the value of the passed parameter is 1.

Example 7-40 SubProc05 subprocedure

```
dcl-Proc subProc05;
  dcl-Pi subProc05;
    action int(10) const;
  end-Pi;

  dcl-S msg1 char(20) inz('Cancel SubProc05 1');
  dcl-S msg2 char(20) inz('Cancel SubProc05 2');
  dcl-S a int(10) inz(20);
  dcl-S b int(10);
  dcl-S c int(10);

  registerCancelStack(%pAddr(cancelStack)
    :%Addr(msg1)
    :*omit);
```

```

registerCancelStack(%pAddr(cancelStack)
                   :%Addr(msg2)
                   :*omit);
// Your code here
if (action = 1);
    endAG(*Omit: *Omit);
endIf;
end-Proc;

```

Note: You can create this program by running the following command:

CRTBNDRPG PGM(REDBOOK/CANCEL02) SRCFILE(REDBOOK/EXCEPTSRC)

Call the program CANCEL01 three times and the following output is displayed:

```

DSPLY      1
DSPLY      2
DSPLY      3
DSPLY  Cancel SubProc04
DSPLY  Cancel SubProc05 2
DSPLY  Cancel SubProc05 1
DSPLY  Cancel SubProc02
DSPLY  AGCancel in CANCEL02
DSPLY  AG ended normally
DSPLY  AG ended by exit request (CEETREC)

```

The third call causes CANCEL02 to be called.

The call to SubProc04 is caused by the divide by zero error.

The final call to SubProc05 results in the AG being ended and all exit procedures being called. starting with the call stack procedures (working back up the call stack) and ending with the AG exit procedures.

7.6.3 Further information

For more information about using the ILE CEE APIs in relation to exception/error handling, see 4.2.6, “Call stack and error handling” on page 121.

You can also refer to the ILE Programming Guide, specifically the Calling Programs and Procedures/Using Bindable APIs and Debugging and Exception Handling/Handling Exceptions sections:

https://www.ibm.com/support/knowledgecenter/ssw_ibm_i_73/rzasc/rzascmain.htm

7.7 Priority of handlers

When an exception/error occurs in a program or a procedure, this is the priority of the error handlers in RPG IV:

1. ‘E’ Operation extender (or Error Indicator)
2. MONITOR group
3. ILE condition handler
4. File exception/error subroutine or program exception/error subroutine (*PSSR)
5. RPG default error handler (which does not apply to subprocedures)

Exceptions/errors are percolated back up the call stack in an ILE environment. You must ensure that you are not inadvertently trapping an unexpected exception/error.

7.8 The embedded SQL problem

When it comes to exception/error handling, embedded SQL assumes you are using **SQLCODE** or **SQLSTATE** to check whether each statement worked, which means that an embedded SQL statement cannot directly cause an RPG program to fail. For RPG file processing, this is the equivalent of having an Error Extender on every file operation code and checking the **%error()** BIF after the operation code.

But, with a little coding, it is possible to get embedded SQL to send a “fail” message to a program and have the normal RPG exception/error handling (*PSSR, monitor group, and so on) handle exception/errors. All that is needed is a call to a subprocedure after every SQL statement.

Example 7-41 shows how the **check_SQLState()** subprocedure is called after a statement. The call to **check_SQLState()** after the second insert statement causes the program to fail because the insert is attempting to insert a duplicate key. The call to **check_SQLState()** after the Fetch shows how the subprocedure can be used to detect an end of file/row not found condition.

Example 7-41 How check_SQLState() subprocedure is called after a statement

```
exec SQL
  insert into empa values(1, 'Paul');
check_SQLState();
exec SQL
  insert into empa values(1, 'Fred');
check_SQLState();
exec sql
  fetch next from C001 into :data ;
if (check_SQLState());
  // It was EOF
endif;
*inLR = *on;
```

Because there is no exception/error in the program, this following output is the resulting job log after a call to **check_SQLState()** fails:

```
Duplicate key value specified.
23505 Duplicate key value specified.
Function check. CPF9897 unmonitored by SQLCATCH at statement 0000003600,
  instruction X'0000'.
The call to check_SQLS ended in error (C G D F)
```

The message that causes the program to fail is that the call to **check_SQLState()** ended in error, but there is an earlier message that shows that a duplicate key value that is specified, followed by the same message that is preceded by the SQLSTATE value (this message is sent by the **check_SQLState()** subprocedure).

So what exactly does the `check_SQLState()` subprocedure do? Based on the value of `SQLSTATE`, it does one of the following actions:

- ▶ Do nothing (because everything is OK).
- ▶ Send a diagnostic message (if there is a warning).
- ▶ Send an escape message if there is a serious error message. The process of sending an escape message means that the subprocedure fails, which means that the caller receives the call to `check_SQLS()` ended in error.

Example 7-42 shows the prototype for the `check_SQLState()` subprocedure, which is coded in the `STDMMSGINFO` member. There are no parameters and it returns a true indicator if an EOF status is detected.

Example 7-42 check_SQLState() subprocedure prototype

```
// Catch and throw an SQL exception/error
dcl-Pr check_SQLState ind extProc('check_SQLState');
end-Pr;
```

`check_SQLState()` is coded in a separate module and should be placed in a service program of your choice - Example 7-43. In order to work, the subprocedure must be in the same activation group as the program.

Example 7-43 check_SQLState() module

```
Ctl-Opt noMain option(*srcStmt: *noDebugIO);

// ***NOTE*** Module must be in the same AG as the caller

/include EXCEPTSRC,STDMMSGINFO

dcl-Proc check_SQLState export;
  dcl-Pi *N ind;
  end-Pi;

// Work fields
  dcl-S messageKey char(4);
  dcl-S messageType char(10);
  dcl-S messageText char(1024);
  dcl-S status ind;

(1)   dcl-Ds *n;
      lastState char(5);
      status_SQL char(2) overLay(lastState);
    end-Ds;

// Constants
  dcl-C W_DIAGNOSTIC  '*DIAG';
  dcl-C W_EOF          '02';
  dcl-C W_ESCAPE        '*ESCAPE';
  dcl-C W_MSGF          'QCPFMMSG  *LIBL';
  dcl-C W_MSGID         'CPF9897';
  dcl-C W_STACK_ENTRY   '*';
  dcl-C W_STACK_COUNT1  1;
  dcl-C W_SUCCESS        '00';
  dcl-C W_WARNING        '01';
```

```

        // Get last state
        exec SQL
(2)      get diagnostics condition 1 :lastState = RETURNED_SQLSTATE;

        // All OK - just return
        if (status_SQL = W_SUCCESS);

        // EOF - return true - but no message
        elseif (status = W_EOF);
(3)      status = *on;

        // Warning - send Diagnostic message
        elseif (status_SQL = W_WARNING);
(4)      messageType = W_DIAGNOSTIC;
        exec SQL
                get diagnostics condition 1 :messageText = MESSAGE_TEXT;

        // Anything else - send an Escape message
        else;
(5)      messageType = W_ESCAPE;
        exec SQL
                get diagnostics condition 1 :messageText = MESSAGE_TEXT;
        status = *on;

        endIf;

        if (messageType <> *blanks);
(6)      messageText = lastState + ' ' + messageText;
        sndPgmMsg( W_MSGID
                  : W_MSGF
                  : messageText
                  : %len(%trimr(messageText))
                  : messageType
                  : W_STACK_ENTRY
                  : W_STACK_COUNT1
                  : messageKey
                  : APIError );
        endif;

        return status;

end-Proc;

```

Note: You can create the SQLCATCH module by running the following command:

CRTRPGMOD MODULE(REDBOOK/SQLCATCH) SRCFILE(REDBOOK/EXCEPTSRC)

Add the module to the service program of your choice.

Note the following points in Example 7-43 on page 300 as indicated by the numbers on the left sides of the examples:

1. A data structure is used to make it easier to extract the first two character of SQLSTATE.

2. **GET DIAGNOSTICS** is used to retrieve the last SQLSTATE. The retrieved value determines what the subprocedure does.
3. If the status indicates a Row Not Found condition, then change the return value to on.
4. If the status indicates a warning message, set the condition to send a diagnostic message.
5. Otherwise, set the condition to send an escape message.
6. If required, send a diagnostic or an escape message.

There are a few other points that you might want to consider.

- ▶ The subprocedure can easily be modified to take into account and handle certain conditions, such as duplicate row or a constraint violation, which be controlled by using an optional parameter.
- ▶ If you prefer to send the your own message instead of re-sending the received message, you can pass the required message text as a parameter.
- ▶ If required, SQLSTATE can be checked before calling the **check_SQLState()** subprocedure. Just be careful of using any SQL that might reset SQLSTATE.
- ▶ If the **check_SQLState()** subprocedure is not going to be in the same activation group as the caller, you can always pass SQLSTATE as a parameter.

7.9 Using percolation: try, throw, and catch

There are a myriad of ways in which programs process application validation and error notification. There are even more methods available when you introduce subprocedures.

The MONITOR group offers the ability to running some operations and catching any exception/errors that might occur. With a couple of simple subprocedures, you can use MONITOR groups as a means of catching and identifying application errors and exception/errors.

As a means of comparison, look at a “normal” means of notifying errors in 7.9.1, “A traditional approach”.

7.9.1 A traditional approach

TRYCATCH01 (Example 7-44) shows an example of a traditional means of performing validation between a buying price and a selling price.

Example 7-44 Source code for TRYCATCH01

```

ctl-Opt dftActGrp(*no) actGrp(*new) option(*srcStmt: *noDebugIO);

dcl-Pr fatalError extPgm('REDBOOK/FATALPGMA');
end-Pr;

dcl-S buy  packed(11:2);
dcl-S sell packed(11:2);
dcl-S msg   char(7);

dsply 'Enter Buying Price:'  '' buy;
dsply 'Enter Selling Price:' '' sell;

msg = validCosts(buy:sell);

```

```

if (msg <> *blanks);
  dsplv msg;
endif;
*inLR = *on;

begSR *PSSR;
  monitor;
    fatalError();
  on-Error;
    dsplv 'Agggghhhhh!';
  endMon;
endSR '*CANCL';

dcl-Proc validCosts;
  dcl-Pi validCosts char(7);
    buyPrice packed(11:2) const;
    sellPrice packed(11:2) const;
  end-Pi;

  dcl-S errMsgId char(7);

  if (buyPrice >= sellPrice);
    errMsgId = 'ERR0001';
  elseIf (sellPrice/buyPrice) > 3 ;
    errMsgId = 'ERR0002';
  endIf;
  return errMsgId;
end-Proc;

```

Note: You can create this program by running the following command:

CRTBNDRPG PGM(REDBOOK/TRYCATCH01) SRCFILE(REDBOOK/EXCEPTSRC)

The program inputs and validates a Buying Price and a Selling Price. The ValidCosts subprocedure performs two validations between the two values passed as parameters; it returns a seven-character message ID indicating the error. The calling procedure checks whether the returned value is blanks and if not, processes any error. Although this is a workable method of notifying errors, it depends on the calling procedure checking the return value from the subprocedure.

Call the program a few times providing valid and invalid values for the buying and selling prices. Note how the standard exception/error handling is invoked if you provide a value of zero for the buying price; it causes a divide by zero error in the ValidCosts subprocedure.

7.9.2 Throw

Instead of simply returning a message ID, you can instigate your own exception/error by having a program or procedure send an escape message. The easiest way to do this is to write a subprocedure to do it, which requires a couple of additions to the STDMMSGINFO copy member, as shown in Example 7-45.

Example 7-45 STDMMSGINFO copy member

```

// Send Message to Program Message Queue
dcl-Pr sndPgmMsg extPgm('QMHSNDPM');

```

```

errorMsgId  char(7)    const;
msgFile     char(20)   const;
msgData     char(3000)  const;
msgDtaLen   int(10)   const;
msgType     char(10)   const;
callStack   char(19)   const;
callStackCtr int(10)   const;
msgKey      char(4);
errorForAPI like(APIError);
end-Pr;

// Throw a message up the call stack
dcl-Pr throw extProc('throw');
  msgId      char(7)    const;
  msgDataIn  varchar(3000) const options(*omit:*noPass);
  msgFileIn  char(20)    const options(*noPass);
end-Pr;

```

SndPgmMsg is a prototype for the Send Program Message (QMHSNDPM) API. QMHSNDPM is called to send an escape message back up the call stack.

Throw is a prototype for a subprocedure that calls the QMHSNDPM API to send a requested message back up the call stack.

The Throw subprocedure is coded in the source member MSGPROCS as shown in Example 7-46.

Example 7-46 Source for Throw subprocedure

```

(1)   ctl-Opt noMain option(*srcStmt: *noDebugIO);

       /include EXCEPTSRC,STDMMSGINFO

       dcl-Proc throw export;
         dcl-Pi throw;
           msgId      char(7)    const;
           msgDataIn  varchar(3000) const options(*omit:*noPass);
           msgFileIn  char(20)    const options(*noPass);
         end-Pi;

(2)   dcl-S msgFile char(20) inz('ERRORS      REDBOOK      ');
       dcl-S msgData char(3000);
       dcl-S msgKey  char(4);

(3)   If (%parms() > 2);
       msgFile = msgFileIn;
     endIf;

     if (%parms() > 1);
       if (%addr(msgDataIn) <> *null);
         msgData = msgDataIn;
       endIf;
     endIf;

(4)   sndPgmMsg( msgId

```

```

        : msgFile
        : msgData
        : %len(%trim(msgData))
        : '*ESCAPE'
        : '*'
        : 1
        : msgKey
        : APIError);
    return;
end-Proc;

```

Note the following points in Example 7-46 on page 304 as indicated by the numbers on the left side of the example:

1. The **NOMAIN** keyword is specified on the H spec because MSGPROCS contains only subprocedures.
2. The default message file is ERRORS in the library REDBOOK (this file may be overridden by the third parameter passed). This message file contains the messages ERR0001 and ERR0002.
3. The subprocedure determines whether optional parameters were passed and overrides work fields accordingly.
4. The SndPgmMsg procedure is called to send the requested error message. The message is sent as an escape message and it is sent to the previous entry in the call stack.

7.9.3 Catch

RPG standard exception/error handling now traps any message that you send (using the Throw procedure) as a 202 status error (Called program or procedure failed). You need another subprocedure to catch the error so you can determine what it is.

Again, this action requires a couple of additions to the STDMMSGINFO copy member, as shown in Example 7-47.

Example 7-47 STDMMSGINFO copy member

```

// Catch a thrown message
dcl-Pr catch extProc('CATCH');
    caughtMessage LikeDS(base_CaughtMessage);
end-Pr;

// Message format returned by Catch
dcl-Ds base_CaughtMessage qualified template;
    msgId   char(7);
    msgFile char(20);
        msgFileName char(10) overLay(msgFile);
        msgLibName  char(10) overLay(msgFile: *next);
    msgText  char(132);
    msgdata  char(3000);
end-Ds;

```

Catch is a prototype for a subprocedure that retrieves a thrown message. Message information is returned in the data structure and passed as a parameter.

Base_CaughtMessage defines the data structure, which is passed as a parameter to Catch. The data structure contains subfields defining the message ID, message file, first level message text and message data for the message retrieved by the Catch subprocedure.

The Catch subprocedure is also coded in the source member MSGPROCS, as shown in Example 7-48.

Example 7-48 MSGPROCS source member

```

dcl-Proc catch export;
  dcl-Pi catch;
    caughtMessage likeDS(base_CaughtMessage);
  end-Pi;

  dcl-Ds msgBack likeDs(RCVM0300) Inz;

  dcl-S setMsgKey char(4) inz(*ALLx'00');

  clear caughtMessage;
receiveMsg( msgBack
  : %size(msgBack)
  : 'RCVM0300'
  : '*'
  : 1
  : '*PRV'
  : setMsgKey
  : 0
  : '*REMOVE'
  : APIError);

  If (msgBack.byteAvail > 0);
    caughtMessage.msgId = msgBack.msgId;
    caughtMessage.msgFileName = msgBack.msgFileName;
    caughtMessage.msgLibName = msgBack.msgLibUsed;
    caughtMessage.msgText =
      %subSt(msgBack.msgData:
        msgBack.lenReplace1 + 1:
        msgBack.lenMsgReturn);
    if (msgBack.lenReplace1 > 0);
      caughtMessage.msgData =
        %subSt(msgBack.msgData:
          1:
          msgBack.lenReplace1);

    endIf;
  endIf;
end-Proc;

```

Note: You can create the MSGPROCS service program by running these commands:

CRTRPGMOD MODULE(REDBOOK/MSGPROCS) SRCFILE(REDBOOK/EXCEPTSRC)
CRTSRVPGM SRVPGM(REDBOOK/MSGPROCS) EXPORT(*ALL)

The binding directory ERROR contains an entry for the MSGPROCS service program.

Catch uses the ReceiveMsg procedure (QMHCRCVPM) to retrieve the last message in the program message queue of the calling program or procedure and places the retrieved information in the parameter data structure.

7.9.4 Using throw and catch

TRYCATCH02 (Example 7-49) shows the implementation of the throw and catch subprocedures.

Example 7-49 Source code for TRYCATCH02

```

ctl-Opt dftActGrp(*no) actGrp(*new) option(*srcStmt: *noDebugIO)
(1)      bndDir('ERROR');

(2) /include EXCEPTSRC,STDMMSGINFO

dcl-Pr fatalError extPgm('REDBOOK/FATALPGMA');
end-Pr;

(3) dcl-Ds caughtMessage LikeDS(Base_CaughtMessage);
dcl-S buy Packed(11:2);
dcl-S sell Packed(11:2);

Dsply 'Enter Buying Price:'    'Buy';
Dsply 'Enter Selling Price:'   'Sell';

(4) monitor;
    validCosts(buy:sell);
on-Error;
(5) catch(caughtMessage);
    dsply caughtMessage.msgId;
endMon;
*inLR = *on;

begSR *PSSR;
monitor;
fatalError();
on-Error;
    dsply 'Agggghhhhh!';
endMon;
endSR '*CANCL';

dcl-Proc validCosts;
dcl-Pi validCosts;
    buyPrice packed(11:2) const;
    sellPrice packed(11:2) const;
end-Pi;

if (buyPrice >= sellPrice);
    throw('ERR0001');
elseIf (sellPrice/buyPrice) > 3 ;
    throw('ERR0002');
endIf;
return;
end-Proc;

```

Note: You can create this program by running the following command:

```
CRTBNDRPG PGM(REDBOOK/TRYCATCH02) SRCFILE(REDBOOK/EXCEPTSRC)
```

Note the following points in Example 7-49 on page 307 as indicated by the numbers on the left side of the example:

1. The ERROR binding directory is specified on the H spec to bind to the Throw and Catch subprocedures in the MSGPROCS service program.
2. The STDMMSGINFO copy member is included for the required prototypes.
3. CaughtMessage is used as the parameter for the Caught subprocedure.
4. The call to ValidCosts is placed in a MONITOR group. Any exception/error causes control to pass to the ON-ERROR. ValidCosts no longer returns a message ID.
5. The Catch procedure is called to determine what the error was.
6. When there is an error, the Throw procedure is called to send the relevant error message and signal an exception/error to the calling program or procedure.

7.9.5 The problem with throw and catch

There are two issues you must be aware of when using throw and catch.

The process of calling the throw subprocedure immediately ends the subprocedure that issues the call. Consider if the ValidateCosts subprocedure were coded as shown in Example 7-50.

Example 7-50 ValidateCosts subprocedure

```
if (buyPrice >= sellPrice);
   throw('ERR0001');
endif;
if (sellPrice/buyPrice) > 3 ;
   throw('ERR0002');
endif;
```

The second validation is not performed if the first validation caused ERR0001 to be thrown. The fact that the Throw procedure sends an escape message means that the call to the Throw procedure ends in error. Because this error is not trapped, the procedure ends immediately.

When you call TRYCATCH02 and provide a value of 0, the divide by zero error is now trapped by the MONITOR and control passes to the ON-ERROR operation as opposed to the exception/error being percolated up the call stack.

You can try to handle these exception/errors after the catch, but it is probably easier to handle them in the Catch subprocedure.

The catch subprocessure can check the message ID that is retrieved, and if it is not an application error (starts with ERR), the message is re-thrown. The code is then added to the catch subprocessure after the contents of the parameter data structure are set. See Example 7-51.

Example 7-51 Code that is added to the Catch subprocessure

```
if (%subSt(caughtMessage.msgId:1:3) <> 'ERR');
  throw( caughtMessage.msgId
        : caughtMessage.msgData
        : caughtMessage.msgFile);
endIf;
```

The problem with re-throwing the error is that the catch subprocessure is now identified as the subprocessure that received the exception/error.

Another alternative is to have the Catch subprocessure call the FatalError procedure. This method requires two changes: the message action parameter on the call to ReceiveMsg in the Catch subprocessure should be *SAME as opposed to *REMOVE, and the Call Stack Counter parameter on the call to ReceiveMsg in the FATALPGMA program should be 3 as opposed to 2 (you want it to retrieve the message from the program message queue of the program or procedure that called Catch).

7.10 Conclusion

The implementation of exception/error handling in your programs is straight forward.

In an environment where you are not using ILE features, such as subprocessures (that is, the programs are running in the default activation group) you can use a combination of the following methods:

- ▶ Write a standard *PSSR subroutine and place it in a copy member.
- ▶ Write a “Fatal Error” program that is called from the *PSSR subroutine.
- ▶ Include the *PSSR subroutine in programs by using a /COPY directive.
- ▶ Include an INFSR(*PSSR) keyword for every file.
- ▶ Implicitly open files.
- ▶ Use Error Extenders and Monitor groups to handle specific exceptions errors.
- ▶ In an environment where you are running programs in ILE activation groups (that is, the programs are not running in the default activation group) you can also write and register your own condition handlers and exit procedures.



Interfacing

One significant area where changes are likely to occur in your RPG programs relate to making it easier to integrate RPG with newer interfaces. XML is already easier to do with RPG. Several of these new interfaces including Java, Python, and PHP are examined in this chapter. The ability to integrate the RPG code that has been reliably serving business application requirements for years is critical to the smooth implementation of these new technologies.

The following interfaces with RPG topics are discussed in this chapter:

- ▶ 8.1, “Interfacing with Java” on page 312
- ▶ 8.2, “Python” on page 335
- ▶ 8.3, “PHP” on page 338

8.1 Interfacing with Java

This section describes RPG IV interfacing with Java.

8.1.1 Java calling RPG IV

There are various methods that can be used to call RPG IV programs and procedures from Java.

The IBM Toolbox for Java contains classes that can be used to call programs or service programs. It also supports the use of Program Call Markup Language (PCML) to describe the program parameters. Because Java also includes support for calling web services, Integrated Web Services (IWS) support can be used to call an RPG IV program or procedure.

Two other techniques that can be used are messaging and stored procedure calls. Because messaging is the recommended technique for calling Java from RPG IV, this topic is described in “LISTOBJSP” on page 312.

Also, because stored procedures are supported by a standard interface and can return lists of data, they are a flexible technique for calling RPG IV from Java. See “Java stored procedure call” on page 313 for more details.

LISTOBJSP

A stored procedure can return values as parameters or as one or more result sets. Because the sample ListObjects API returns a list of values, you can create a simple wrapper program that calls the API and returns the object information in a result set as shown in Example 8-1.

After you have the program, you can expose it as a stored procedure by using an **SQL CREATE PROCEDURE** call, as described in the LISTOBJSP program comments.

Example 8-1 Source for LISTOBJSP stored procedure

```
// Stored procedure wrapper program for our API

// Since the SQL precompiler needs to see the data structure in the
// listobjpr copy, we need to specify the RPGPOPT(*LVL1) option
// on the CRTSQLRPGI command

// The SQL to create the stored procedure (using system naming) is:
//
// create procedure radredbook/listobjsp(
//   in objname char(10), in objlib char(10), in objtype char(10))
//   result sets 1 language rpgle deterministic contains sql
//   external name radredbook/listobjsp parameter style general

// Use our API binding directory
ctl-opt main(main) dftactgrp(*no) bnddir('MYAPI');

// Include the prototypes using /copy since they are needed by
// the SQL preprocessor
/copy listobjpr

// Main procedure
dcl-proc main;
  dcl-pi main extpgm('LISTOBJSP');
```

```

Name      char(10) const;
Library   char(10) const;
Type      char(10) const;
end-pi;

dcl-ds ObjectInfo likeds(ObjectInfo_t) dim(1000);
dcl-s  count int(10);
dcl-s  rc    int(10);

rc = ListObjects(Name:Library>Type:count:ObjectInfo);
if rc = 0;
  exec sql set result sets for return to client array :ObjectInfo
    for :count rows;
endif;
end-proc;

```

Java stored procedure call

The stored procedure can be called from Java by using the standard SQL interface, as shown in Example 8-2.

Example 8-2 Source for calling the stored procedure

```

public void run()
{
  try
  {
    DriverManager.registerDriver(new com.ibm.as400.access.AS400JDBCDriver());
    Connection connection = DriverManager.getConnection("jdbc:as400://" +
      + system, getJDBCProperties());
    PreparedStatement statement = connection
      .prepareStatement("CALL LISTOBJSP(?, ?, ?)");
    statement.setString(1, "L*");
    statement.setString(2, "RADREDBOOK");
    statement.setString(3, "*SRVPGM");
    ResultSet rs = statement.executeQuery();
    while (rs.next())
    {
      System.out.println(rs.getString("LIBRARY") + "/" + rs.getString("NAME") +
        + " " + rs.getString("TYPE") + " " + rs.getString("ATTRIBUTE") +
        + " " + rs.getString("DESCRIPTION"));
    }
    statement.close();
    connection.close();
  }
  catch (SQLException e)
  {
    e.printStackTrace();
  }
}

```

The output of this method is shown in Example 8-3.

Example 8-3 Output from the Java call

RADREDBOOK/LISTOBJ	*SRVPGM	RPGLE	Example list objects API
--------------------	---------	-------	--------------------------

8.1.2 RPG IV calling Java

Although calling RPG IV from Java can be done in various simple ways, calls in the other direction are not as easy. Although RPG IV contains built-in support for accessing Java objects and methods, there are various problems with the use of that support. Because that support results in a JVM for each job and as a result you are likely to have problems with scalability.

A preferred approach is to start a server in a JVM that can process requests in different threads and send messages to that server. This approach scales much better because you have only one JVM running. The messaging can be done with data queues, IBM MQ, or sockets.

We create a socket-based example that allows RPG IV to retrieve information from a spreadsheet that is stored in the Integrated File System. The Apache Software Foundation has a project that is called POI that contains a set of Java classes that can be used to work with Microsoft documents. We create a Java server that can process requests for information in a spreadsheet and the RPG IV routines to send those requests and receive the responses.

SocketDispatcher

Java has a relatively high-level socket interface, so it is not difficult to create a flexible multi-threaded socket server as shown in Example 8-4. The server listens on a specified port on the TCP loopback address so that it is available only to local client connections. When a socket connection request is accepted, the socket is passed to a processor that is specific to the application.

Example 8-4 Source for the SocketDispatcher program

```
package com.richdiedrich.redbook;

import java.io.IOException;
import java.net.*;
/**
 *
 * Simple Java socket server to handle message based requests.
 * The server will only listen for connections on the loopback address.
 *
 */
public class SocketDispatcher
{
    private int port;
    private int backlog = 50;
    private Class<? extends SocketProcessor> processclass;

    /**
     * Create a dispatcher on a specified port for a specified processor class.
     *
     * @param port the port number for this service
     * @param processclass a subclass of SocketProcessor
    
```

```
        */
    public SocketDispatcher(int port, Class<? extends SocketProcessor> processclass)
    {
        this.port = port;
        this.processclass = processclass;
    }

    /**
     * Main processing loop
     *
     */
    public void run()
    {
        ServerSocket ss = null;
        try
        {
            ss = new ServerSocket(port, backlog, InetAddress.getLoopbackAddress());
            for (;;)
            {
                Socket s = ss.accept();
                try
                {
                    SocketProcessor processor = processclass.newInstance();
                    processor.setSocket(s);
                    new Thread(processor).start();
                }
                catch (Exception e)
                {
                    e.printStackTrace();
                }
            }
            catch (IOException e)
            {
                e.printStackTrace();
            }
        finally
        {
            try
            {
                if (ss != null)
                    ss.close();
            }
            catch (Exception e)
            {
            }
        }
    }

    /**
     * Set the maximum server backlog.
     * The default value is 50.
     *
     * @param backlog maximum number of queued requests.
     */
}
```

```
public void setBacklog(int backlog)
{
    this.backlog = backlog;
}
```

SocketProcessor

The `SocketProcessor` class provides the code that is required to receive a request message, call the `processRequest` method, and send a response message, as shown in Example 8-5. Because the actual request processing depends on the specific application, the `processRequest` method is not implemented in this class, but is implemented in an application-specific subclass.

Example 8-5 Source code for `SocketProcessor` class

```
package com.richdiedrich.redbook;

import java.io.*;
import java.net.Socket;

/**
 *
 * Abstract class for processing socket messages
 *
 */
public abstract class SocketProcessor implements Runnable
{
    private Socket s = null;
    private final String FS = "\u001F";

    /**
     *
     * Set the socket for request and response messages.
     *
     * @param s the connected socket.
     */
    public void setSocket(Socket s)
    {
        this.s = s;
    }

    /**
     *
     * The actual request processing method
     *
     * @param request array of strings making up request and parameters
     * @return array of response strings
     * @throws Exception
     */
    abstract public String[] processRequest(String[] request) throws Exception;

    public void run()
    {
        DataInputStream in = null;
        DataOutputStream out = null;
```

```

try
{
    in = new DataInputStream(s.getInputStream());
    out = new DataOutputStream(s.getOutputStream());
    for (;;)
    {
        byte[] request = new byte[in.readInt()];
        in.readFully(request);
        String[] respvalues = processRequest(splitMessage(new String(request,
            "UTF8")));
        if (respvalues != null)
        {
            byte[] response = prefixMessage("Response", buildMessage(respvalues))
                .getBytes("UTF8");
            out.writeInt(response.length);
            out.write(response);
        }
    }
}
catch (EOFException e)
{
}
catch (Exception e)
{
    try
    {
        byte[] error = prefixMessage("Error", e.toString()).getBytes("UTF8");
        out.writeInt(error.length);
        out.write(error);
    }
    catch (Exception e1)
    {
    }
}
finally
{
    try
    {
        s.close();
    }
    catch (Exception e)
    {
    }
}
}

/**
 * Prepend a prefix to a message string.
 *
 * @param prefix value to prepend.
 * @param message original message string.
 * @return concatenated strings.
 */
private String prefixMessage(String prefix, String message)
{

```

```

        StringBuilder builder = new StringBuilder(prefix);
        builder.append(FS);
        builder.append(message);
        return builder.toString();
    }

    /**
     * Join an array of strings into a single string using a Unicode delimiter
     * character.
     * @param in array of strings to be joined.
     * @return delimited string.
     */
    private String buildMessage(String[] in)
    {
        StringBuilder builder = new StringBuilder();
        for (String s : in)
        {
            builder.append(s);
            builder.append(FS);
        }
        if (builder.length() > 0)
            builder.setLength(builder.length() - 1);
        return builder.toString();
    }

    /**
     * Split a string using a Unicode delimiter character.
     * @param in delimited string.
     * @return array of string components.
     */
    private String[] splitMessage(String in)
    {
        return in.split(FS);
    }
}

```

SpreadsheetSocketProcessor

The SpreadsheetSocketProcessor class contains the processRequest method that is needed to implement an application-specific set of functions. This class uses the Apache Software Foundation POI support to retrieve some information from an .xlsx spreadsheet as shown in Example 8-6.

Example 8-6 Source code for SpreadsheetSocketProcessor

```

package com.richdiedrich.redbook;

import java.io.IOException;

import org.apache.poi.xssf.usermodel.XSSFSheet;
import org.apache.poi.xssf.usermodel.XSSFWorkbook;

/**
 *
 * SocketProcessor for retrieving information from a spreadsheet in IFS
 *

```

```

* The requests are:
*
* sheetnames: return a list of the sheet names from a spreadsheet
* getColumn: return a list of string column values from a spreadsheet
*
*/
public class SpreadsheetSocketProcessor extends SocketProcessor
{
    private String filename = null;
    private XSSFWorbook workbook = null;

    public String[] processRequest(String[] request) throws Exception
    {
        if (request.length == 0)
            throw new Exception("No request data");
        if (request[0].equalsIgnoreCase("sheetnames"))
        {
            if (request.length < 2)
                throw new Exception("Missing parameter");
            readFile(request[1]);
            String[] names = new String[workbook.getNumberOfSheets()];
            for (int i = 0; i < names.length; i++)
            {
                names[i] = workbook.getSheetAt(i).getSheetName();
            }
            return names;
        }
        else if (request[0].equalsIgnoreCase("getColumn"))
        {
            if (request.length < 4)
                throw new Exception("Missing parameter");
            int column = Integer.parseInt(request[3]);
            readFile(request[1]);
            XSSFSheet sheet = workbook.getSheetAt(Integer.parseInt(request[2]));
            String[] columns = new String[sheet.getLastRowNum() + 1];
            for (int i = 0; i < columns.length; i++)
            {
                columns[i] = sheet.getRow(i).getCell(column).getStringCellValue();
            }
            return columns;
        }
        return null;
    }

    /**
     * Read a file from IFS into a workbook for processing.
     * @param filename IFS path to spreadsheet file.
     * @throws IOException
     */
    private void readFile(String filename) throws IOException
    {
        if (filename.equalsIgnoreCase(this.filename) && this.workbook != null)
            return;

        this.filename = filename;
    }
}

```

```

        workbook = new XSSFWorkbook(filename);
    }
}

```

SpreadSheetServer

The SpreadSheetServer class is used to start a SocketDispatcher on the appropriate port with the application-specific processing class as shown in Example 8-7.

Example 8-7 Source for SpreadSheetServer

```

/**
 * Simple wrapper to start a specific SocketDispatcher.
 */
public class SpreadSheetServer
{
    static final int PORT = 5678;
    public static void main(String[] args)
    {
        SocketDispatcher dispatcher = new SocketDispatcher(PORT,
            SpreadsheetSocketProcessor.class);
        dispatcher.run();
    }
}

```

SOCKPR

Just as the example Java code was built in layers with the application-specific code running on top of general-purpose routines, the example RPG IV code is also built in layers.

Although the QSYSINC library contains RPG IV members for some of the structures and prototypes that are needed, it does not currently contain a full set of the prototypes that are necessary for socket communication. The SOCKPR file contains the needed definitions, as shown in Example 8-8.

Example 8-8 SOCKPR definitions

```

/IF DEFINED(SOCKPR_Included)
  /EOF
/ENDIF
/DEFINE SOCKPR_Included

/include QSYSINC/QRPGLESRC,SYSTYPES
/include QSYSINC/QRPGLESRC,ERRNO

// 
// System socket definitions and prototypes
//

dcl-c AF_INET          2; // Internet domain
dcl-c SOCK_STREAM       1; // Stream
dcl-c SO_KEEPALIVE      25; // Keep alive flag
dcl-c SOL_SOCKET         -1; // Socket level option
dcl-c TCP_NODELAY        10; // Do not delay sending small packets
dcl-c IPPROTO_TCP        6; // Transmission control protocol

dcl-ds in_addr           qualified template;

```

```

    s_addr      like(u_long);
end-ds;

dcl-s sa_family_t like(u_short) template;
dcl-s in_port_t  like(u_short) template;

dcl-ds in6_addr      qualified template;
    u6_addr32   like(u_long) dim(4);
    u6_addr16   like(u_short) dim(8) pos(1);
    u6_addr8    like(u_char) dim(16) pos(1);
end-ds;

dcl-ds sockaddr_in    qualified template;
    sin_family  like(i_short);
    sin_port    like(u_short);
    sin_addr    likeds(in_addr);
    sin_zero    like(u_char) dim(8);
end-ds;

dcl-ds sockaddr_in6   qualified template;
    sin6_family like(sa_family_t);
    sin6_port   like(in_port_t);
    sin6_addr   likeds(in6_addr);
    sin6_scope_id like(u_long);
end-ds;

dcl-ds hostent      qualified template;
    h_name      pointer;
    h_aliases   pointer;
    h_addrtype  like(i_int);
    h_length    like(i_int);
    h_addr_list pointer;
end-ds;

dcl-ds h_addr_list   qualified template;
    address     pointer;
end-ds;

dcl-ds timeval       qualified template;
    tv_sec      like(i_long);
    tv_usec     like(i_long);
end-ds;

dcl-ds fdset         qualified template;
    fdes        like(u_int) dim(7);
end-ds;

dcl-pr socket        like(i_int) extproc('socket');
    addr_family like(i_int) value;
    type        like(i_int) value;
    protocol    like(i_int) value;
end-pr;

dcl-pr inet_addr     like(u_long) extproc('inet_addr');
    addr_string  pointer value options(*string:*trim);

```

```

end-pr;

dcl-pr setsockopt like(i_int) extproc('setsockopt');
  sock_desc    like(i_int) value;
  level       like(i_int) value;
  option_name  like(i_int) value;
  option_value  pointer   value;
  option_length like(i_int) value;
end-pr;

dcl-pr gethostbyname pointer extproc('gethostbyname');
  host_name     pointer value options(*string:*trim);
end-pr;

dcl-pr connect   like(i_int) extproc('connect');
  sock_desc    like(i_int) value;
  dest_addr    pointer value;
  addr_len     like(i_int) value;
end-pr;

dcl-pr select    like(i_int) extproc('select');
  max_desc     like(i_int) value;
  read_set      likeds(fdset) options(*omit);
  write_set     likeds(fdset) options(*omit);
  except_set    likeds(fdset) options(*omit);
  wait_time     likeds(timeval) options(*omit);
end-pr;

dcl-pr send      like(i_int) extproc('send');
  sock_desc    like(i_int) value;
  buffer       pointer value;
  buffer_length like(i_int) value;
  flags        like(i_int) value;
end-pr;

dcl-pr recv      like(i_int) extproc('recv');
  sock_desc    like(i_int) value;
  buffer       pointer value;
  buffer_length like(i_int) value;
  flags        like(i_int) value;
end-pr;

dcl-pr close     like(i_int) extproc('close');
  sock_desc    like(i_int) value;
end-pr;

```

JAVAMSGPR

The first example module contains the procedures that are needed for communicating with the Java `SocketDispatcher` and `SocketProcessor` classes. It does not contain the procedures specific to the application. The module is built from two source members. The first contains the declarations that are used by procedures that use the interface, and the second contains the actual implementation of the procedures.

The declarations are shown in Example 8-9.

Example 8-9 Procedures declarations and procedure implementation

```
// Data definitions

dcl-s Message_t varchar(10000) ccsid(*utf8) template;
dcl-c MAXSPLITS 2000;
dcl-c FS          x'1F';      // Unicode delimiter character

// Receive a message
//
// Input parameters:
//   connection - an open socket connection
//   message - buffer to receive message
//
// Returns:
//   Return code - 0 or a value indicating the error
dcl-pr recvMessage int(10);
  connection int(10) value;
  message    like(Message_t);
end-pr;

// Send a message
//
// Input parameters:
//   connection - an open socket connection
//   message - message to be sent
//
// Returns:
//   Return code - 0 or a value indicating the error
dcl-pr sendMessage int(10);
  connection int(10) value;
  message    like(Message_t) value ccsid(*UTF8);
end-pr;

// Connect to socket based Java service
//
// Input parameter:
//   port - port number for service
//
// Returns:
//   connection - connection value or -1 for error
dcl-pr connectJVM int(10);
  port int(10) value;
end-pr;

// Disconnect from socket based Java service
//
// Input parameter:
//   connection - open socket connection
dcl-pr disconnectJVM;
  connection int(10) value;
end-pr;

// Access socket errno value
```

```

// Input parameter:
// newval - if present, sets errno to this value
//
// Returns:
// errno value
dcl-pr errno int(10);
    newval int(10) options(*nopass);
end-pr;

// Generates list of split points for a delimited string
//
// Input parameters:
// instring - string to be split
// maxsplit - maximum number of splits to be returned
// splitsfound - number of split sections found
// splits - array of integers containing split points
// splitchar - if present overrides the default of x'1F'
//
// Returns:
// Return code - 0 or a value indicating the error
dcl-pr splitString int(10);
    instring like(Message_t) const options(*varszie);
    maxsplit int(10) value;
    splitsfound int(10);
    splits int(10) dim(MAXSPLITS) options(*varszie);
    splitchar char(1) options(*nopass) ccsid(*utf8);
end-pr;

// Return a specific split from a string
//
// Input parameters:
// split - the split number
// instring - string that was split using splitString
// splits - array of integers containing split points from splitString
//
// Returns:
// Return code - 0 or a value indicating the error
dcl-pr getSplit like(Message_t) rtnparm;
    split int(10) value;
    instring like(Message_t) const options(*varszie);
    splits int(10) dim(MAXSPLITS) options(*varszie);
end-pr;

```

The implementation of JAVAMSG is show in Example 8-10.

Example 8-10 Source for JAVAMSG

```

// Routines for processing socket based requests to a Java service
// process.
//
// See JAVAMSGPR for descriptions of exported procedures

ctl-opt nomain;

/include javamsgpr

```

```

/include sockpr

dcl-proc recvMessage export;
  dcl-pi recvMessage int(10);
    connection int(10) value;
    message    like(Message_t);
  end-pi;

  dcl-s length int(10);

  // Get length value
  if recvFully(connection:%addr(length):%size(length)) <> 0;
    return -1;
  endif;
  // Incoming message is too long
  if length > %len(message):*max);
    return -2;
  endif;
  // Set the length value of the return parameter
  %len(message) = length;
  // Receive message data
  if recvFully(connection:%addr(message:*data):length) <> 0;
    return -1;
  endif;
  return 0;
end-proc;

dcl-proc sendMessage export;
  dcl-pi sendMessage int(10);
    connection int(10) value;
    message    like(Message_t) value ccsid(*UTF8);
  end-pi;

  dcl-s rc int(10);
  dcl-s length int(10);

  // Send the message length
  length = %len(message);
  rc = send(connection:%addr(length):%size(length):0);
  // Send the message data
  if rc = %size(length);
    rc = send(connection:%addr(message:*data):length:0);
  endif;
  if rc = length;
    rc = 0;
  endif;
  return rc;
end-proc;

dcl-proc connectJVM export;
  dcl-pi connectJVM int(10);
    port int(10) value;
  end-pi;
  dcl-s connection int(10);
  dcl-ds address likeds(sockaddr_in);

```

```

dcl-s rc int(10);
dcl-s nodelay int(10);

// Get a TCP socket
connection = socket(AF_INET:SOCK_STREAM:IPPROTO_TCP);
if connection = -1;
  return connection;
endif;

// Set no delay option on socket
nodelay = 1;
rc = setsockopt(connection:IPPROTO_TCP:TCP_NODELAY:%addr(nodelay):
                 %size(nodelay));

// Loopback address
address = *allx'00';
address.sin_family = AF_INET;
address.sin_port = port;
address.sin_addr.s_addr = inet_addr('127.0.0.1');

// Connect
if connect(connection:%addr(address):%size(address)) = -1;
  rc = close(connection);
  return -1;
endif;

return connection;
end-proc;

dcl-proc disconnectJVM export;
dcl-pi disconnectJVM;
  connection int(10) value;
end-pi;
dcl-s rc int(10);
rc = close(connection);
end-proc;

dcl-proc splitString export;
dcl-pi splitString int(10);
  instring like(Message_t) const options(*varszie);
  maxsplit int(10) value;
  splitsfound int(10);
  splits int(10) dim(MAXSPLITS) options(*varszie);
  splitchar char(1) options(*nopass) ccsid(*utf8);
end-pi;
dcl-s pos int(10) inz(0);
dcl-s c char(1) inz(FS) ccsid(*utf8);

// Change split character
if %parms >= %parmnum(splitchar);
  c = splitchar;
endif;

// Look through string for split character
splitsfound = 0;

```

```

dow splitsfound < maxsplit;
    splitsfound += 1;
    if pos < %len(instring);
        pos = %scan(c:instring:pos + 1);
        if pos > 0;
            splits(splitsfound) = pos;
        else;
            splits(splitsfound) = %len(instring) + 1;
            leave;
        endif;
    else;
        splits(splitsfound) = %len(instring) + 1;
        leave;
    endif;
enddo;
if splitsfound > 0 and splits(splitsfound) < %len(instring);
    return -1;
endif;
return 0;
end-proc;

dcl-proc getSplit export;
    dcl-pi getSplit like(Message_t) rtnparm;
        split      int(10) value;
        instring   like(Message_t) const options(*varsize);
        splits     int(10) dim(MAXSPLITS) options(*varsize);
    end-pi;
    if split = 1;
        return %subst(instring:1:splits(split) - 1);
    endif;
    if splits(split - 1) < %len(instring);
        return %subst(instring:splits(split - 1) + 1:
                           splits(split) - splits(split - 1) - 1);
    endif;
    return '';
end-proc;

dcl-proc errno export;
    dcl-pi errno int(10);
        newval int(10) options(*nopass);
    end-pi;
    dcl-s errnoval int(10) based(errnoptr);
    errnoptr = getErrnoPtr();
    if %parms >= %parmnum(newval);
        errnoval = newval;
    endif;
    return errnoval;
end-proc;

// A single socket receive may not always return the full length
// requested
// This routine will loop until full buffer size is received
dcl-proc recvFully;
    dcl-pi recvFully int(10);
        connection int(10) value;

```

```

        buffer      pointer value;
        size       int(10) value;
end-pi;
dcl-s length int(10);
dow size > 0;
    length = recv(connection:buffer:size:0);
    if length <= 0;
        return -1;
    endif;
    buffer += length;
    size -= length;
endo;
return 0;
end-proc;

```

XLSXUTILPR

After you have the procedures that are required to communicate with the Java code, you can create the procedures specific to the example application. Again, this module is built from an interface source member and an implementation source member.

Example 8-11 show the code for the interface source member.

Example 8-11 Source code for XLSXUTILPR

```

// Open a connection to spreadsheet server JVM
//
// Returns:
// Return code - 0 or a value indicating the error
dcl-pr openXLSXConnection int(10);
end-pr;

// Close connection to JVM
//
// Input parameters:
// connection - an open socket connection
dcl-pr closeXLSXConnection;
    connection int(10) value;
end-pr;

// Get the sheet names from a spreadsheet in IFS
//
// Input parameters:
// path - IFS path for spreadsheet
// provided - number of elements provided in names array
// available - number of elements returned in names array
// names - array of strings to contain sheet names
// curconnection - an open socket connection, if not provided a new
//                  connection will be opened and closed for this
//                  request
//
// Returns:
// Return code - 0 or a value indicating the error
dcl-pr getXLSXSheets int(10);
    path      varchar(200) value;
    provided  int(10) value;

```

```

available int(10);
names    varchar(100) dim(100) options(*varsize);
curconnection int(10) value options(*nopass);
end-pr;

// Get the string values for a column in a spreadsheet
//
// Input parameters:
// path - IFS path for spreadsheet
// sheet - the sheet name
// column - the column number
// provided - number of elements provided in columns array
// available - number of elements returned in columns array
// names - array of strings to contain column values
// curconnection - an open socket connection, if not provided a new
//                  connection will be opened and closed for this
//                  request
//
// Returns:
// Return code - 0 or a value indicating the error
dcl-pr getXLSXColumn int(10);
path      varchar(200) value;
sheet     int(10) value;
column    int(10) value;
provided   int(10) value;
available  int(10);
columns   varchar(100) dim(100) options(*varsize);
curconnection int(10) value options(*nopass);
end-pr;

// Get more information on the last error returned
//
// Returns:
// Java exception value or socket errno value
dcl-pr getXLSXError varchar(100);
end-pr;

```

XLSXUTIL

Example 8-12 show the code for the interface source member.

Example 8-12 Source code for XLSXUTIL

```

// Routines for IFS spreadsheet requests
//
// See XLSXUTILPR for descriptions of exported procedures
ctl-opt nomain;

/include xlsxutilpr
/include javamsgpr

dcl-c JVMPORT 5678;           // Port for this service
dcl-s lasterror varchar(100);  // Details on most recent error

dcl-proc openXLSXConnection export;

```

```
dcl-pi openXLSXConnection int(10);
end-pi;
dcl-s connection int(10);
connection = connectJVM(JVMPORT);
if connection = -1;
  lasterror = %char(errno());
endif;
return connection;
end-proc;

dcl-proc closeXLSXConnection export;
  dcl-pi closeXLSXConnection;
    connection int(10) value;
  end-pi;
  disconnectJVM(connection);
end-proc;

dcl-proc getXLSXSheets export;
  dcl-pi getXLSXSheets int(10);
    path      varchar(200) value;
    provided  int(10) value;
    available int(10);
    names     varchar(100) dim(100) options(*varsize);
    curconnection int(10) value options(*nopass);
  end-pi;

  dcl-s connection int(10);
  dcl-s response  like(Message_t);
  dcl-s rc        int(10);
  dcl-s splits    int(10) dim(101);
  dcl-s splitcount int(10);
  dcl-s respvalue varchar(20);
  dcl-s i         int(10);

// If not current connection, open a new connection
if %parms >= %parmnum(curconnection);
  connection = curconnection;
else;
  connection = openXLSXConnection();
endif;
if connection < 0;
  return -1;
endif;
// Request consists of request name and IFS path
rc = sendMessage(connection:'sheetnames' + FS + %trim(path));
if rc = 0;
  rc = recvMessage(connection:response);
  if rc = 0;
    rc = splitString(response:%elem(splits):splitcount:splits);
    respvalue = getSplit(1:response:splits); // First split
    if respvalue = 'Response'; // If successful,
      splitcount -= 1; // fill names array
      available = splitcount;
      if splitcount > provided;
        splitcount = provided;
```

```

        endif;
        for i = 1 to splitcount;
            names(i) = getSplit(i + 1:response:splits);
        endfor;
    else;                                // If error,
        lasterror = getSplit(2:response:splits); // Store error message
        rc = -2;
    endif;
    else;
        lasterror = %char(errno());           // Store socket error
    endif;
else;
    lasterror = %char(errno());
endif;
if %parms < %parmnum(curconnection); // If new connection, close it
    disconnectJVM(connection);
endif;
return rc;
end-proc;

dcl-proc getXLSXColumn export;
    dcl-pi getXLSXColumn int(10);
    path      varchar(200) value;
    sheet     int(10) value;
    column    int(10) value;
    provided   int(10) value;
    available  int(10);
    rows      varchar(100) dim(100) options(*varszie);
    curconnection int(10) value options(*nopass);
end-pi;

dcl-s connection int(10);
dcl-s response  like(Message_t);
dcl-s rc         int(10);
dcl-s splits    int(10) dim(101);
dcl-s splitcount int(10);
dcl-s respvalue varchar(20);
dcl-s i          int(10);

if %parms >= %parmnum(curconnection);
    connection = curconnection;
else;
    connection = openXLSXConnection();
endif;
if connection < 0;
    return -1;
endif;
rc = sendMessage(connection:'getcolumn' + FS + %trim(path) + FS +
                 %char(sheet - 1) + FS + %char(column - 1));
if rc = 0;
    rc = recvMessage(connection:response);
if rc = 0;
    rc = splitString(response:%elem(splits):splitcount:splits);
    respvalue = getSplit(1:response:splits);
    if respvalue = 'Response';

```

```

        splitcount -= 1;
        available = splitcount;
        if splitcount > provided;
            splitcount = provided;
        endif;
        for i = 1 to splitcount;
            rows(i) = getSplit(i + 1:response:splits);
        endfor;
    else;
        lasterror = getSplit(2:response:splits);
        rc = -2;
    endif;
    else;
        lasterror = %char(errno());
    endif;
    else;
        lasterror = %char(errno());
    endif;
    if %parms < %parmnum(curconnection);
        disconnectJVM(connection);
    endif;
    return rc;
end-proc;

dcl-proc getXLSXError export;
    dcl-pi getXLSXError varchar(100);
end-pi;
return lasterror;
end-proc;

```

TESTXLSX

Now that you have the example application-specific interface written, you can use it in an application. Example 8-13 shows a test program for the spreadsheet Java service.

Example 8-13 Source code for TESTXLSX

```

// Test program for spreadsheet Java service

ctl-opt main(main) dftactgrp(*no) bnkdir('XLSXUTIL');

/include xlxutilpr

dcl-proc main;
    dcl-pi main extpgm('TESTXLSX');
end-pi;

SingleRequest();
MultipleRequest();
end-proc;

// Simple single request
dcl-proc SingleRequest;
    dcl-pi SingleRequest;
end-pi;

```

```

dcl-s sheetnames varchar(100) dim(20);
dcl-s rc int(10);
dcl-s available int(10);
dcl-s i int(10);

display('Single Request');
rc = getXLSXSheets('/radredbook/test1.xlsx':
                     %elem(sheetnames):available:sheetnames);
if rc = 0;
  for i = 1 to available;
    display(sheetnames(i));
  endfor;
else;
  display(getXLSXError());
endif;
end-proc;

// Multiple requests on the same connection
dcl-proc MultipleRequest;
  dcl-pi MultipleRequest;
  end-pi;

  dcl-s sheetnames varchar(100) dim(20);
  dcl-s customers varchar(100) dim(20);
  dcl-s rc int(10);
  dcl-s available int(10);
  dcl-s i int(10);
  dcl-s connection int(10);

  display('Multiple Request');
  connection = openXLSXConnection(); // Open connection for requests
  rc = getXLSXSheets('/radredbook/test1.xlsx':
                     %elem(sheetnames):available:sheetnames:
                     connection); // Use open connection
if rc = 0;
  // Show name of first sheet
  display(sheetnames(1));
  // Get data from second column of first sheet
  rc = getXLSXColumn('/radredbook/test1.xlsx':1:2:
                     %elem(customers):available:customers:
                     connection); // Use open connection
  if rc = 0;
    for i = 1 to available;
      display(customers(i));
    endfor;
  else;
    display(getXLSXError());
  endif;
else;
  display(getXLSXError());
endif;
closeXLSXConnection(connection); // We need to close the connection
end-proc;

// Simple procedure to make strings displayable

```

```
dcl-proc display;
  dcl-pi display;
    string varchar(100) value;
  end-pi;
  dcl-s dsplayval varchar(52);
  dsplayval = string;
  dsply dsplayval;
end-proc;
```

Figure 8-1 shows the spreadsheet output from TESTXLSX.

	A	B	C	D	E	F	G	H	I	J
1	Date	Customer	Hours							
2	01/20/16	Blue Sun Corporation	6							
3	01/27/16	Omni Consumer Products	4							
4	01/28/16	CHOAM	4							
5	01/28/16	Adipose Industries	2							
6										
7										
8										
9										
10										

Figure 8-1 Resulting spreadsheet from TESTXLSX

Here is the output of the test program:

```
DSPLY Single Request
DSPLY January
DSPLY February
DSPLY March
DSPLY April
DSPLY May
DSPLY June
DSPLY July
DSPLY August
DSPLY September
DSPLY October
DSPLY November
DSPLY December
DSPLY Multiple Request
DSPLY January
DSPLY Customer
DSPLY Blue Sun Corporation
DSPLY Omni Consumer Products
DSPLY CHOAM
DSPLY Adipose Industries
```

8.1.3 Things to remember

Here are some important points to remember:

- ▶ Java calling RPG IV
 - IBM Toolbox for Java classes:
 - Call programs or procedures.
 - PCML describes the interfaces.
 - Web Services:

- Can call Integrated Web Services interfaces.
 - WSDL describes the interface.
 - Messaging:
 - Data queue
 - Socket
 - MQ
 - Messages must be defined
 - Stored procedure:
 - Standard interface
 - Might require a wrapper program
 - An **SQL CREATE PROCEDURE** is required
 - Can return lists of results
- The system handles most of the interface details.
- ▶ RPG IV calling Java
 - Using RPG IV direct call support is not recommended.
 - Messaging is a useful scalable technique:
 - The message-format processing should be designed for reuse.
 - The Java server process should be multi-threaded to handle simultaneous requests.
 - The Java server process must be running.

8.2 Python

Just as with Java, there are a variety of methods that can be used to call RPG IV from Python. In addition to the standard stored procedure and web services calls, there is the Toolkit for IBM i that works with the XMLSERVICE library to provide access to IBM i resources.

The XMLSERVICE library is available from the Young i Professionals website:

<http://yips.idevcloud.com/wiki/index.php/XMLService/XMLService>

8.2.1 Calling a stored procedure

The LISTOBJJSP stored procedure that was shown in “LISTOBJJSP” on page 312 can also be called using the Python ibm_db package.

Example 8-14 shows the Python source for calling the stored procedure.

Example 8-14 Python source for calling a stored procedure

```
# Import database module
import ibm_db
# Import module containing userid and password
import dbccred

def list_objects(library, objtype = '*ALL', object = '*ALL'):
    # Connect to database
    conn = ibm_db.connect('*LOCAL', dbccred.userid, dbccred.password)
    # Prepare a statement with parameter markers
    stmt = ibm_db.prepare(conn, 'CALL RADREDBOOK.LISTOBJJSP(?, ?, ?)')
    # Set parameters
```

```

ibm_db.bind_param(stmt, 1, object, ibm_db.SQL_PARAM_INPUT)
ibm_db.bind_param(stmt, 2, library, ibm_db.SQL_PARAM_INPUT)
ibm_db.bind_param(stmt, 3, objtype, ibm_db.SQL_PARAM_INPUT)
# Execute statement
ibm_db.execute(stmt)
# Build response
objectlist = []
row = ibm_db.fetch_assoc(stmt)
while row != False:
    objectlist.append(row)
    row = ibm_db.fetch_assoc(stmt)
# Close connection
ibm_db.close(conn)
# Return response
return objectlist

```

Toolkit for IBM i

In addition to building the Python language for IBM i, a toolkit was also created to help integrate the ability to call IBM i objects. The tool kit leverages the XML Services support, but helps the developer from having to write all the XML. The toolkit handles the interaction between the Python code and the IBM i.

Example 8-15 shows calling the ListObject API directly leveraging the toolkit.

Example 8-15 Source for calling ListObject API with the toolkit

```

# Import from itoolkit modules
from itoolkit.lib.ilibcall import *
from itoolkit import *

def i_list_objects(library, objtype = '*ALL', object = '*ALL'):
    # Initialize transport
    itransport = iLibCall()

    # New service call
    itool = iToolKit()
    # Add set library list
    itool.add(iCmd('chglbl', 'CHGLIBL LIBL(RADREDBOOK)'))
    # Add service program call
    itool.add(
        # Input parameters
        iSrvPgm('listobjects', 'LISTOBJ', 'LISTOBJECTS')
            .addParm(iData('objectname', '10a', object))
            .addParm(iData('library', '10a', library))
            .addParm(iData('objtype', '10a', objtype))
        # Output parameters - returncount contains number of array elements with data
        .addParm(iData('returncount', '10i0', '', {'enddo':'rtncount'}))
        .addParm(
            iDS('objectinfo', {'dim':'1000', 'dou':'rtncount'})
                .addData(iData('NAME', '10a', ''))
                .addData(iData('LIBRARY', '10a', ''))
                .addData(iData('TYPE', '10a', ''))
                .addData(iData('ATTRIBUTE', '10a\'v '))
                .addData(iData('DESCRIPTION', '50a\'v '))
        )
    )

```

```

        .addRet(iData('rc', '10i0', ''))
    )

    # Perform service call
    itool.call(itransport)
    # Get the result of the service program call
    listobjout = itool.dict_out('listobjects')
    # If it worked, display results
    if 'success' in listobjout:
        # Get the output records
        return listobjout['objectinfo']
    else:
        return null

```

Test script

The test script shown in Example 8-16 shows how to designed the APIs using the same interface even if they are called using two different methods.

Example 8-16 Test script to call the APIs

```

#!/usr/bin/env python3
import listobjects
import ilistobjects

# Run the stored procedure version
objectlist = listobjects.list_objects('RADREDBOOK', '*ALL', 'L*')
for object in objectlist:
    print(object['NAME'].ljust(11) + object['TYPE'].ljust(11) +
          object['DESCRIPTION'])

# Run the toolkit version
iobjectlist = ilistobjects.i_list_objects('RADREDBOOK', '*ALL', 'L*')
for iobject in iobjectlist:
    print(iobject['NAME'].ljust(11) + iobject['TYPE'].ljust(11) +
          iobject['DESCRIPTION'])

```

Output

Example 8-17 shows the output results after running the test script in Example 8-16.

Example 8-17 Output results of running the test script

LISTOBJSP *PGM	List objects stored procedure
LISTOBJTST *PGM	List objects test program
LISTOBJ *SRVPGM	Example list objects API
LISTOBJ *MODULE	Example list objects API
LISTOBJSP *PGM	List objects stored procedure
LISTOBJTST *PGM	List objects test program
LISTOBJ *SRVPGM	Example list objects API
LISTOBJ *MODULE	Example list objects API

8.3 PHP

The same techniques that were used with Python can also be used with PHP.

There is also a PHP toolkit for IBM i that simplifies the accessing of IBM i ILE objects, programs, and data. Since PHP has been available on IBM i for nearly a decade now, there are many examples and helps available from many sources including IBM RedBooks, articles, and other open source material. As long as programs are written as procedures that represent stateless callable routines, it is easy to call them in any language.

The IBM Redbooks publication *Modernizing IBM i Applications from the Database up to the User Interface and Everything in Between*, SG24-8185 has a great chapter on PHP with several examples leveraging RPG.

Also reference the Zend IBM i Solutions website. Zend is a strategic partner when it comes to PHP. They have many articles and resources that can help you become successful with PHP on IBM i.

<http://www.zend.com/en/solutions/modernize-ibm-i>



IBM Rational Developer for IBM i

An essential part of developing modern applications with RPG is the toolset that you use when writing, compiling, and debugging the code. The only development toolset from IBM that fully supports free-format RPG is the Rational Developer for IBM i (RDi).

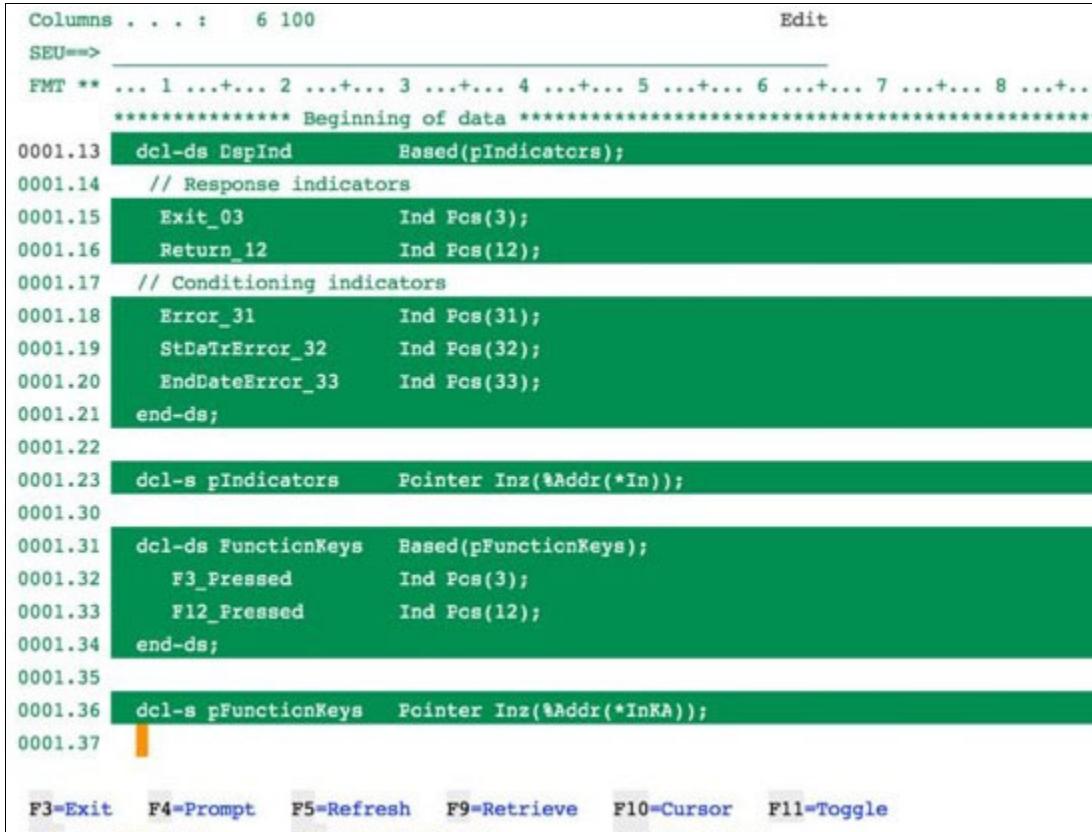
The following RDi topics are covered in this chapter:

- ▶ 9.1, “Why you should use Rational Developer for IBM i” on page 340
- ▶ 9.2, “Built on Eclipse” on page 348
- ▶ 9.3, “What is new in Rational Developer for IBM i” on page 352
- ▶ 9.4, “Using the RDi debugger” on page 360

9.1 Why you should use Rational Developer for IBM i

The older Source Entry Utility (SEU) editor has not been updated for the latest in RPG language capability since IBM i 6.1. SEU indicates a syntax error when a developer uses any keywords for RPG open access or uses new built-in functions such as %ScanRpl (scan and replace). The lack of support for the free-format declarations (that is, the replacements for the H, F, D, and P specs) can hinder productivity in SEU. RDi understands the new declaration syntax and can help supply the keywords that are necessary to use it through its Content Assist feature.

Figure 9-1 and Figure 9-2 on page 341 show the contrast between a window full of errors returned by SEU for valid modern RPG syntax and RDi support.



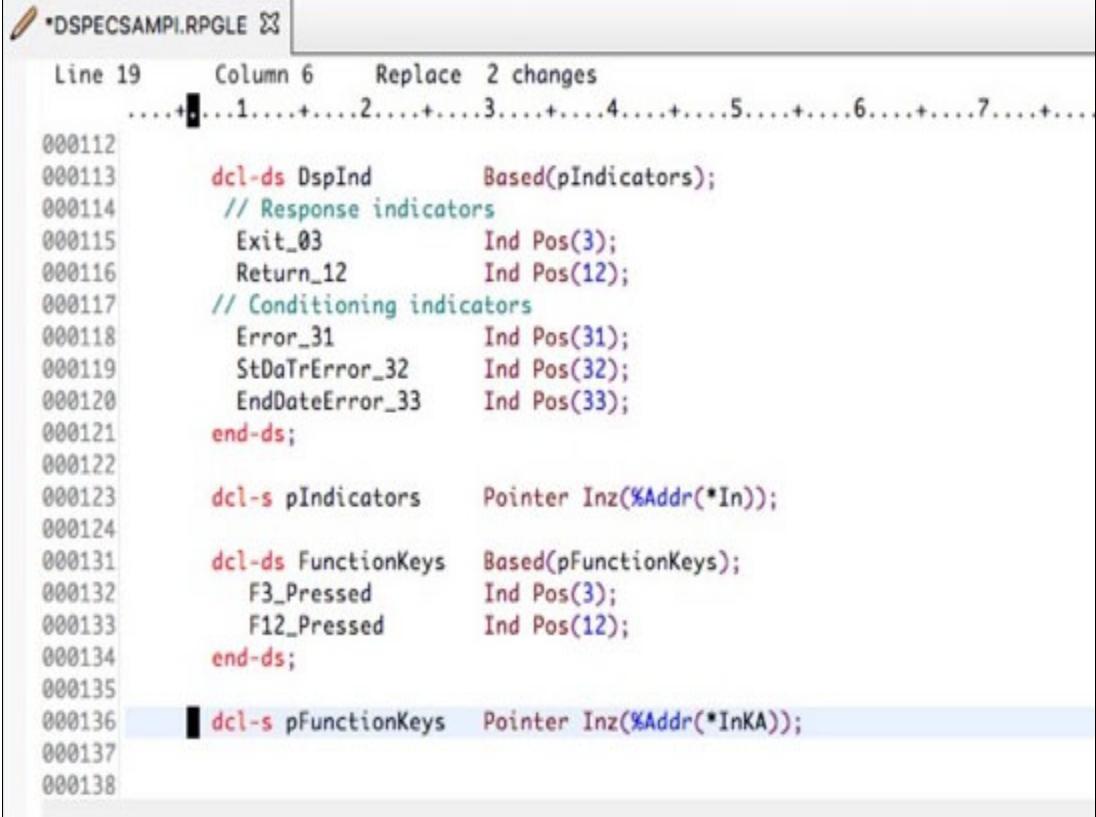
```

Columns . . . :   6 100
SEU==>
FMT ** ... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8 ...+...
***** Beginning of data *****
0001.13 dcl-ds DspInd      Based(pIndicators);
0001.14 // Response indicators
0001.15 Exit_03           Ind Pos(3);
0001.16 Return_12          Ind Pos(12);
0001.17 // Conditioning indicators
0001.18 Error_31           Ind Pos(31);
0001.19 StDaTrError_32     Ind Pos(32);
0001.20 EndDateError_33     Ind Pos(33);
0001.21 end-ds;
0001.22
0001.23 dcl-s pIndicators  Pointer Inz(%Addr(*In));
0001.30
0001.31 dcl-ds FunctionKeys Based(pFunctionKeys);
0001.32 F3_Pressed         Ind Pos(3);
0001.33 F12_Pressed        Ind Pos(12);
0001.34 end-ds;
0001.35
0001.36 dcl-s pFunctionKeys Pointer Inz(%Addr(*InKA));
0001.37

F3=Exit  F4=Prompt  F5=Refresh  F9=Retrieve  F10=Cursor  F11=Toggle

```

Figure 9-1 SEU errors when using modern free-form RPG



```

*DSPECSAMP1.RPGLE ✎
Line 19   Column 6   Replace 2 changes
....+1....+2....+3....+4....+5....+6....+7....+...
000112
000113      dcl-ds DspInd      Based(pIndicators);
000114          // Response indicators
000115          Exit_03      Ind Pos(3);
000116          Return_12     Ind Pos(12);
000117          // Conditioning indicators
000118          Error_31      Ind Pos(31);
000119          StDaTrError_32  Ind Pos(32);
000120          EndDateError_33 Ind Pos(33);
000121      end-ds;
000122
000123      dcl-s pIndicators  Pointer Inz(%Addr(*In));
000124
000125      dcl-ds FunctionKeys Based(pFunctionKeys);
000126          F3_Pressed    Ind Pos(3);
000127          F12_Pressed   Ind Pos(12);
000128      end-ds;
000129
000130      dcl-s pFunctionKeys Pointer Inz(%Addr(*InKA));
000131
000132

```

Figure 9-2 RDi fully supports modern RPG

Because RDi is the only environment that understands the latest in RPG syntax, it is no longer advisable to use the 5250 emulation (“green screen”) tools of Programming Development Manager (PDM) and Source Entry Utility (SEU). Its been well over a decade since any enhancements have been added to these development tools.

The fact that new language functions are not supported in the 5250 emulation editors is not the only reason to move to RDi. In fact, it is not even the best reason. The best reasons are those that enhance productivity. There are too many of those reasons to cover them all here, but this section highlights a few of the biggest productivity-related differences between RDi and PDM/SEU.

This section covers some of the productivity features in RDi compared to using PDM and SEU. It is targeted primarily to those who are not already using RDi. Section 9.3, “What is new in Rational Developer for IBM i” on page 352 covers some more advanced features, such as the use of plug-ins, and some of the more recent enhancements to RDi and 9.4, “Using the RDi debugger” on page 360 discusses debugging RPG programs with RDi i.

The following RDi topics are covered in this section:

- ▶ 9.1.1, “Integrated compile-time error feedback” on page 342
- ▶ 9.1.2, “Outlining your program” on page 342
- ▶ 9.1.3, “Editing multiple members at once” on page 344
- ▶ 9.1.4, “Modern editor capabilities” on page 345
- ▶ 9.1.5, “Moving from SEU/PDM to RDi” on page 347

9.1.1 Integrated compile-time error feedback

In RDi, errors that occur during a compile appear in an error list that is adjacent to the source editor. Double-clicking an error in this list takes you to the line of code in the editor where the error was found. RDi developers do not need to look at separate spool file compile listings to find and fix compile-time errors in their code.

Source members that are edited in RDi are typically compiled without closing the source member to enable a faster reaction to any compile-time errors. As shown in Figure 9-3, a compile attempt resulted in several errors. You simply have to double-clicked on one of the errors and the editor is automatically positioned to the line of code for that error.

The integration of the error feedback with the editor is considerably faster than browsing a spooled file compile listing to find lines of code in error.

ID	Message	Severity	Line	Location	Connection
RNS9308	Compilation stopped. Severity 30 errors found in program.	50	0	RSEEX1/QRPGLESRC(GETDESCRFR)	SID System
RNF7503	Expression contains an operand that is not defined.	30	19	RSEEX1/QRPGLESRC(GETDESCRFR)	SID System
RNF7030	The name or indicator CATCODE is not defined.	30	19	RSEEX1/QRPGLESRC(GETDESCRFR)	SID System
RNF7503	Expression contains an operand that is not defined.	30	25	RSEEX1/QRPGLESRC(GETDESCRFR)	SID System
RNF7030	The name or indicator PRODUCT... is not defined.	30	25	RSEEX1/QRPGLESRC(GETDESCRFR)	SID System

Figure 9-3 RDi integrated compile-time error feedback

9.1.2 Outlining your program

A program outline view is available to help with navigating through the source code. The outline of an RPG IV program includes details of every file that is declared, including all record formats and all fields and their definitions. In addition, the details of all the program-defined variables and data structures are included. In addition to data definitions,

the RPG IV outline view contains all the subroutines, subprocedures, and prototypes that are defined in (or whose definitions are copied into) the source member.

There is far less need for opening other source members to find what parameters must be passed to a program or procedure, or to use commands such as Display File Field Descriptions (**DSFFD**) to get details about externally defined data. Much of the data that is available through the outline is also available in other, even more convenient ways. For example, when you position your cursor on the name of a variable within the source, a definition of it appears as hover text on the screen.

The outline is integrated with the editor, which means that you can navigate directly to the place in the program where a variable or subroutine or other item is defined or coded. Simply click on the line in the outline view and the code area is auto positioned to that line of code.

In addition, there is a cross-reference in the outline showing where each item is referenced in the source member. Those cross-reference lines are also connected to the editor.

In the case of variables, the outline also indicates which lines of code modify the value of the variable versus those that simply reference it. For example, using the outline view, you can navigate to a specific subroutine or procedure or navigate directly to every line of code in the member that calls the subroutine.

If your outline is large, the ability to filter by the name of variables, procedure, or subroutines is helpful. The ability to hide unreferenced items in the outline is particularly helpful when many prototypes have been copied into the program from a single source member, it reduces the prototype list to only those actually used in this program.

Figure 9-4 shows the use of the outline view with an RPG IV program.

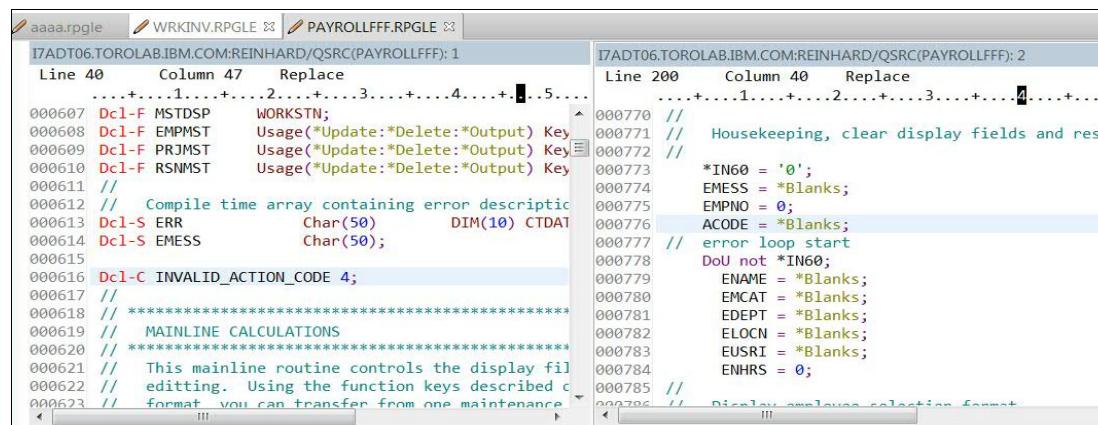
The screenshot shows the Rational Developer for IBM i interface with two main windows. On the left is the code editor window titled 'DSPPRODFR2.RPGLE' showing RPG IV code. The code includes various RPG statements like Open, If, Do, EndIf, EndDo, and EndSR. A specific line of code, 'Ctr = Ctr + 1;', is highlighted with a blue selection bar. On the right is the 'Outline' view window, which displays a hierarchical list of symbols and their locations in the code. A filter bar at the top of the outline window says 'type Filter text'. Below it, a tree view shows categories like 'Main Procedure', 'Subroutines', and 'ClearSubfile'. Under 'Main Procedure', the line '103 (M)' is selected and highlighted in blue. Other entries include '103', '125 (M)', '125', 'Constants', 'Indicators', and 'FillByPrCode'. The 'Main Procedure' node has a count of '28' under it.

Figure 9-4 RPG outline view with filtering

9.1.3 Editing multiple members at once

Today's more modular coding styles mean that code between several source members interact with each other more often. RDi can have multiple source members open for editing or browsing concurrently. It is simply a matter of clicking between the open members, because the logic flows between modules.

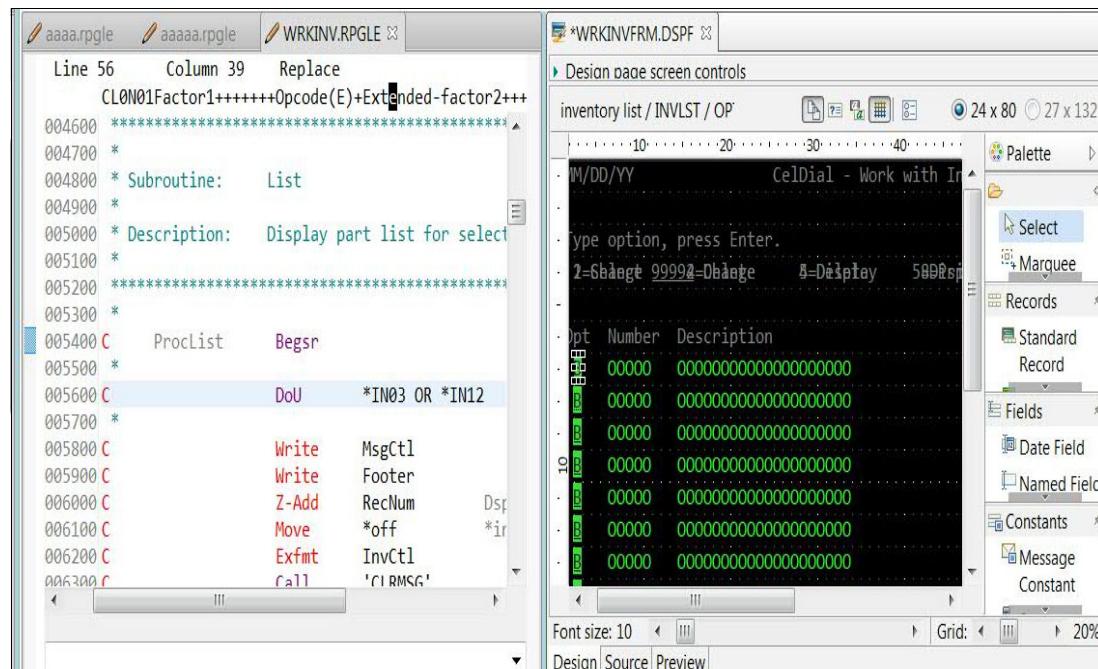
Two or more members can be seen simultaneously through one or more horizontal or vertical split-screen lines. Even when in split-screen mode, all members can be open for editing. It is even possible to split a single member into two views with both sides editable, for example, data declarations open on one side with the logic open beside it. Figure 9-5 shows RDi with the declarations and the code logic in two views.



The screenshot shows two separate windows of the RDi editor. The left window is titled 'aaaa.rpgle' and contains code for member I7ADT06.TOROLAB.IBM.COM:REINHARD/QSRC(PAYROLLFFF):1. The right window is titled 'WRKINV.RPGLE' and contains code for member I7ADT06.TOROLAB.IBM.COM:REINHARD/QSRC(PAYROLLFFF):2. Both windows show RPG code with various declarations, comments, and logic. The windows are separated by a vertical scroll bar.

Figure 9-5 RDi showing code declarations in one view and code logic in another

RDi allows a variety of different possibilities when it comes to the screens that can be displayed at one time. Figure 9-6 shows another example, source on one side and the DDS screen editor on another.



The screenshot shows two windows side-by-side. The left window is titled 'aaaa.rpgle' and displays RPG source code for member aaaa.rpgle. The right window is titled 'WRKINVFRM.DSPF' and displays the DDS (Design Screen) editor for a screen named 'INVLST / OP'. The DDS editor shows a list of fields and their definitions, along with various design tools and palettes on the right side.

Figure 9-6 RDi with source and DDS design editor

9.1.4 Modern editor capabilities

Freed from the 5250 emulation text-based size limitations, the editor in RDi offers many features that could not be implemented in the SEU interface:

- ▶ More code visible at a time

When editing source code, you can typically see two to three times the number of lines of code compared to the SEU editor. When using SEU, the number of lines that is visible is fixed, regardless of the size of monitor that is used. With the RDi editor, the number of source lines visible varies based on the monitor size and shape and the font size that is selected. Seeing more code makes it easier and faster to follow the flow of logic.

- ▶ Undo and redo

During an edit session, you have unlimited levels of both undo and redo of individual changes, even after a save and compile, while the source member is still open in the editor. You can open a file, make changes, save, and then compile, repeat this sequence multiple times, and still undo individual changes all the way back to when you opened that file. In the SEU editor, the only option to undo changes is to undo all the changes in the entire editing session by closing the editor without saving the member.

- ▶ Color coding

With RDi you can assign separate colors to many features of the RPG language. This allows you to instantly recognize something without ever having to think about it. By default, RDi sets comments to the color teal, numbers blue, logic (like 'IF, ELSE) is assigned purple. This simple color coding improves your accuracy as you write code, as the editor instantly assigns color to help you eliminate mistakes.

Figure 9-7 on page 346 shows the color preferences page where you can set the color you want for each aspect of the RPG language.

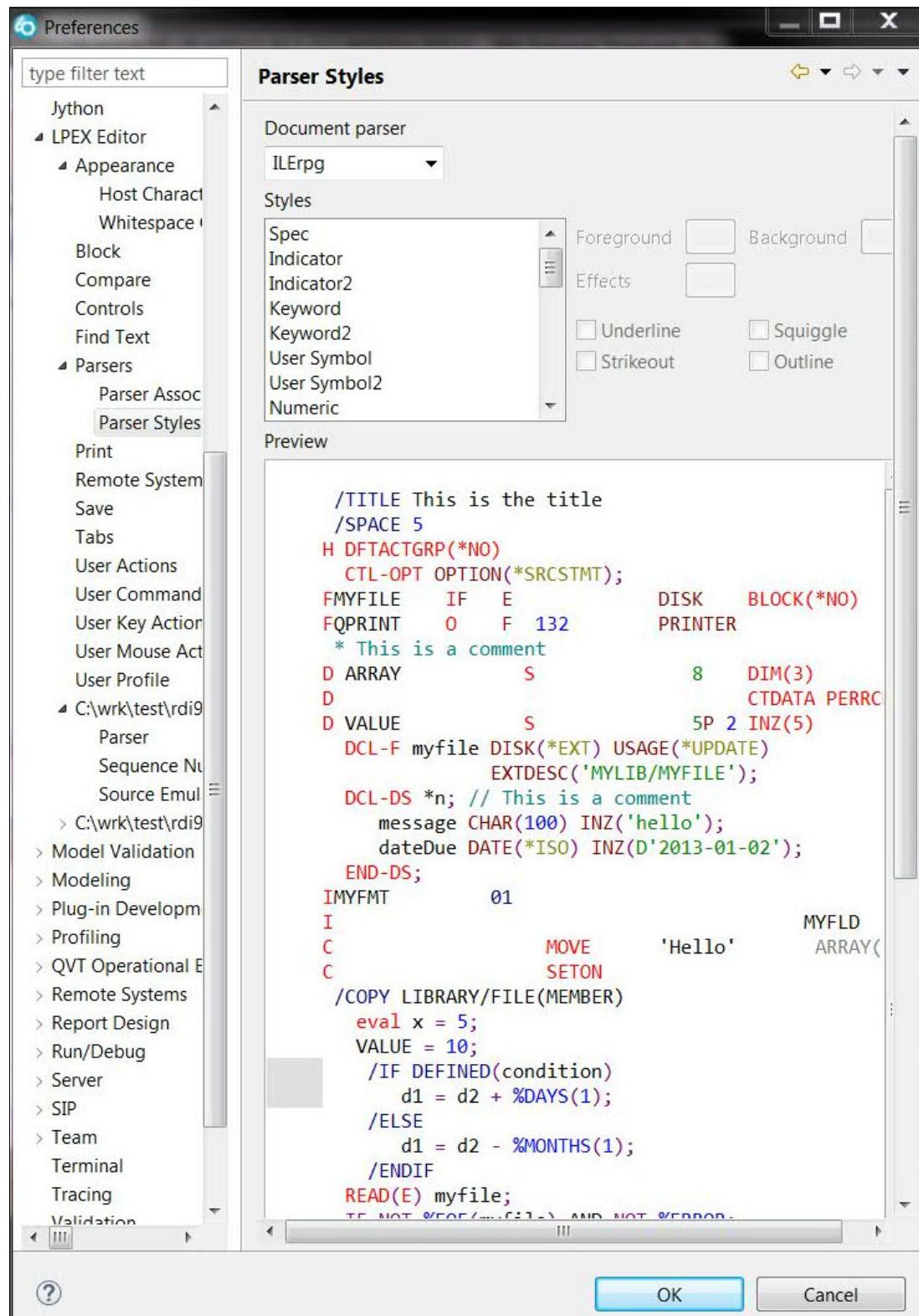


Figure 9-7 Color preferences pane in RDI

- ▶ Content assist

Content assist in RDi can make suggestions to finish code after you start it by taking the context of the cursor position into account. You can start content assist by pressing Ctrl+Space bar.

If you need help remembering the name of an RPG built-in function, type % and press Ctrl+Space bar to open a list of them.

Content assist can provide a list of variable names that are of the appropriate type for a parameter for the built-in function or even for a program or procedure call. If the parameter must be a character, the list contains only character variable names. If you begin to type the first few letters, the list is filtered to show only the appropriate variables that begin with those letters, which is convenient for qualified data structure subfields.

Having trouble remembering what all those new keywords are in the new free-format declarations? After DCLF and the file name, for example, Ctrl+Space bar brings up a list of file declaration keywords.

Figure 9-8 shows the use of content assist with DCLF.

```

Dcl-F Product;
Dcl-DS ProductRec LikeRec(ProductR);
Dcl-C obsoleteProduct '0';
Dcl-S prodRecCount Int(5);

// Definition and variables for report file

Dcl-F ProdRptF Printer
  Dcl-S fullPage
    If fullPage;
      PrintHeaders(
        EndIf;

      Dcl-Proc PrintHe
        Write Heading;
        fullPage = "Of
      End-Proc;

```

Figure 9-8 Content assist with free-form declarations

9.1.5 Moving from SEU/PDM to RDI

A 60-day trial of RDi is available to be used in learning to make the switch from 5250 emulation development tools to RDi. Training on how to best utilize the tools is suggested for those making the transition. Otherwise you might not be aware of many of the features and how to use them.

There are many resources that can be used to help in the learning process. There are a number of companies who offer on-site training in person. In addition the following is a small sampling of the online resources that can prove useful:

- ▶ System i Developer's RSE Quick Start Guide
<http://systemideveloper.com/downloadRSEQuickStartGuide.html>
- ▶ IBM Rational Developer for i website for downloading a trial version:
<https://www.ibm.com/developerworks/downloads/r/rdi/>

- ▶ RDi online learning resources from IBM:
https://ibm.biz/rdi_wiki_self_learning
- ▶ System i Developer favorite shortcut keys card
<http://systemideveloper.com/downloadRSEShortcuts.html>

9.2 Built on Eclipse

RDi is built on the Eclipse Integrated Development Environment (IDE). RDi adds the intelligence to interact with IBM i components, such as libraries, objects, source members, and IFS files. It also adds language support for IBM i languages, such as RPG, COBOL, CL, C, C++, SQL and DDS, for editing and compiling.

An advantage of the Eclipse base is that many other tools are also built on that same base. The skills that you use to edit your RPG and CL code become transferable to coding in other environments, such as PHP or Java.

An even more significant advantage is that the Eclipse IDE explicitly enables plug-ins to be built by anyone. Although IBM has written a set of plug-ins that is packaged as RDi, other companies and individuals are free to develop additional plug-ins that help support IBM i development environments. Most source control and change management software vendors with products in the IBM i marketplace, for example, provide plug-ins to interact with their software.

Some resourceful RDi users create more generic tools that are unrelated to a specific product on the host to make their own development easier and more productive. There are many examples of tools that integrate with RDi. One of the most prolific group of RDi plug-in developers are the creators of iSphere, which is an open source project with a group of extensions to RDi.

9.2.1 iSphere tools

The iSphere developers add new features frequently. This section includes a sampling of some of the iSphere tools that are useful:

- ▶ Finding and editing messages.

Dealing with messages can be cumbersome when using CL commands. iSphere makes it easy with three tools:

- Message file search tool
- Message editor
- Message file compare and merge

You can use the search tool to find the messages that need attention and then edit the message content. Using the compare and merge tool, messages from one message file can be merged into another. Anyone who has experience searching for text in messages and modifying message text using only CL commands will appreciate how much easier the iSphere suite of tools make those tasks.

Figure 9-9 on page 349 show the search capabilities.

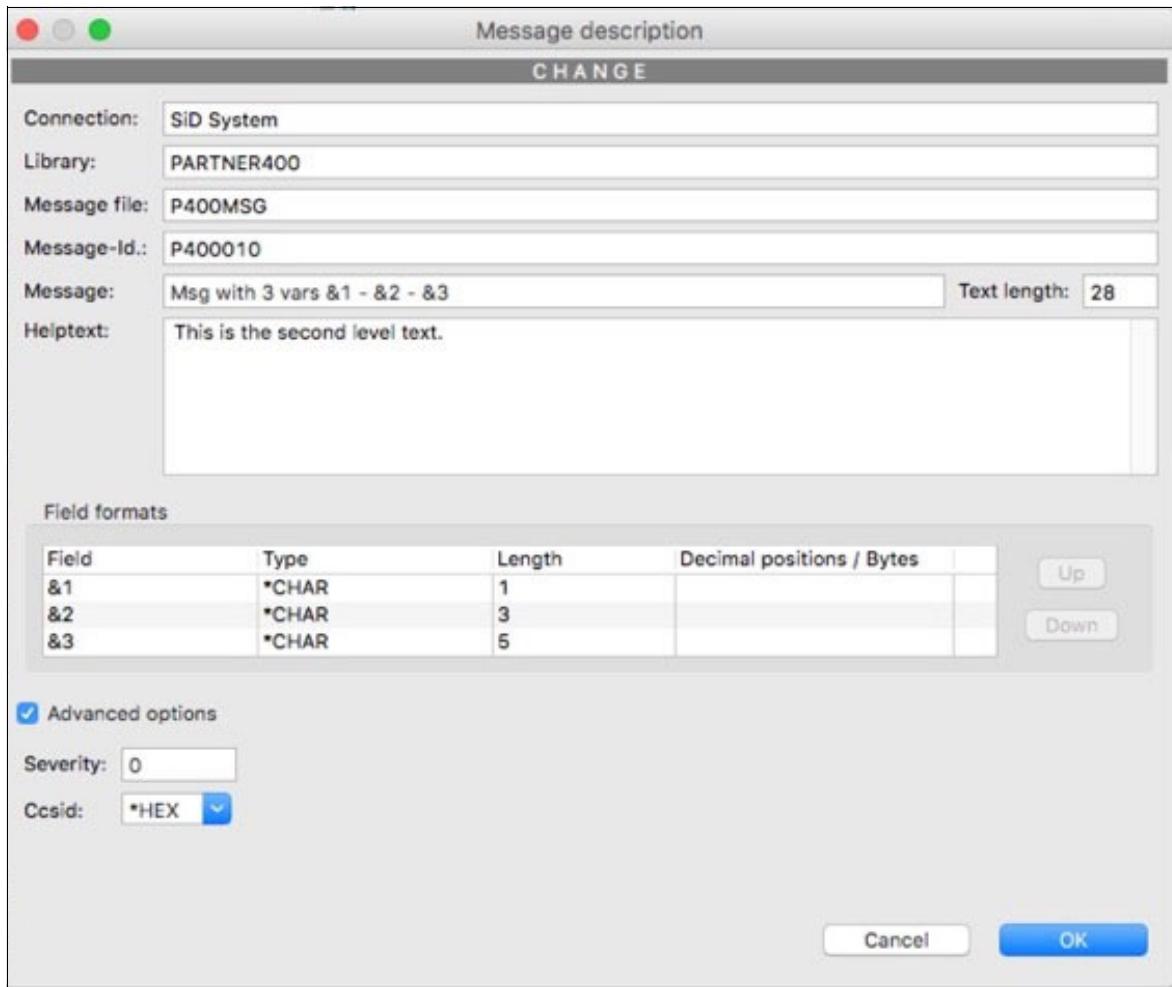


Figure 9-9 iSphere message editor

- ▶ Find strings in source files/members

The base RDi package comes with the ability to search source members from one or more source files on IBM i. The iSphere source search adds additional functions by allowing for multiple search strings, including support for regular expressions.

Figure 9-10 on page 350 illustrates the iSphere Source File Search dialog.

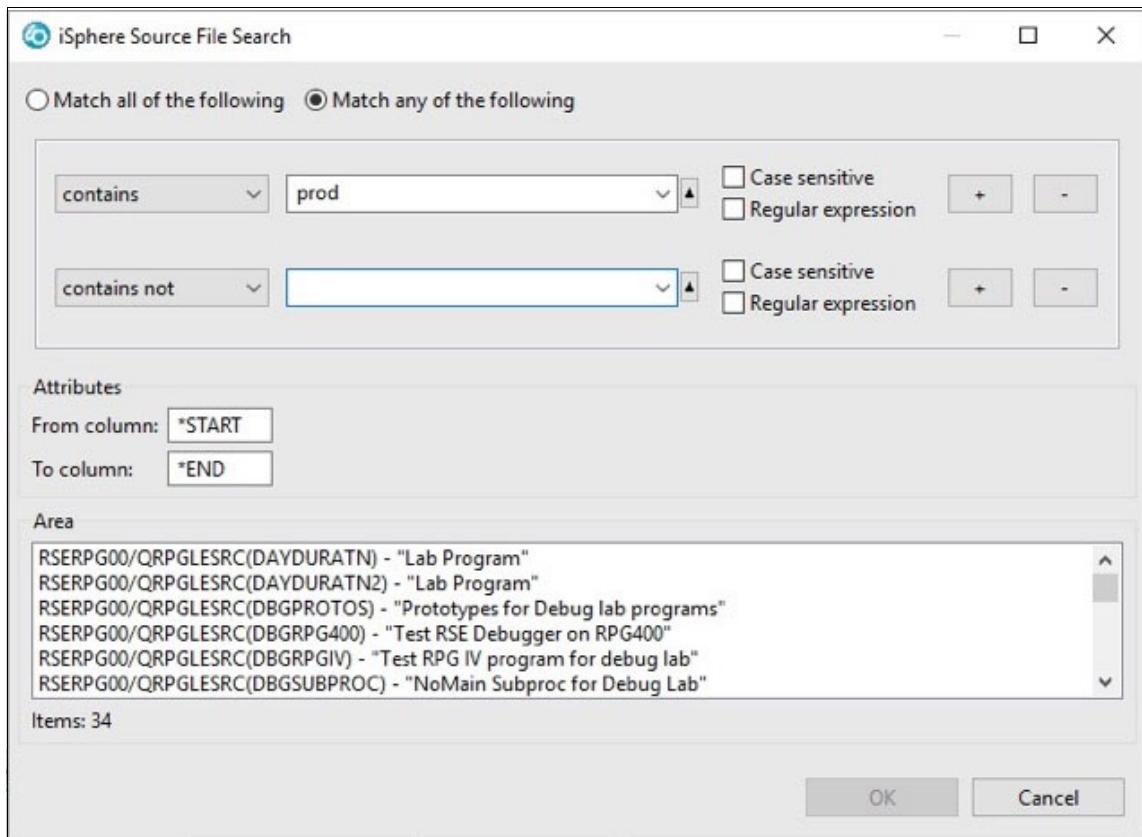


Figure 9-10 iSphere source file search

An iSphere enhancement to the base search capability is that the results from multiple searches can be retained. In addition, the search results can be saved into a spreadsheet, or the members in the search results can be used to create a member filter.

To navigate through the search results, select a source member in the left column. Each statement containing the search strings appears to the right. Double-click a statement to open the source member by using your choice of edit or browse mode.

Each tab that is shown in Figure 9-11 contains the search results from previous searches.

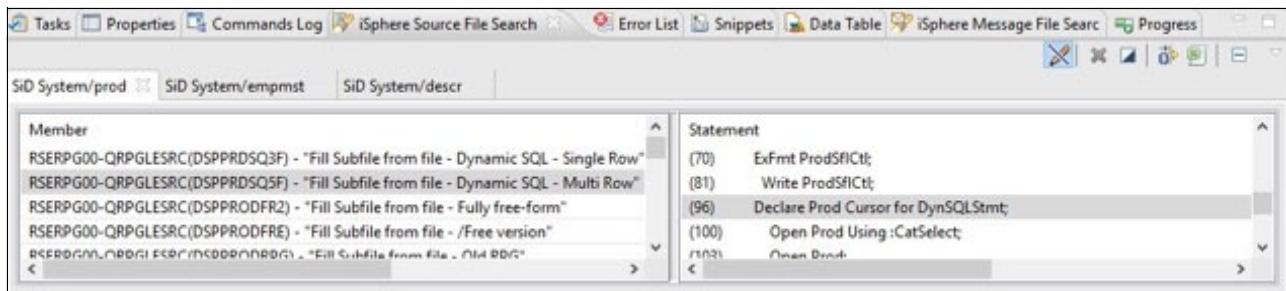


Figure 9-11 iSphere source search results

- ▶ Source/object text decorators

In the base RDi package, if you want to see the text that is associated with a source member or an object, you must use the Table view. iSphere offers a function called *decorators* that shows the text in the Remote Systems view.

Figure 9-12 shows an example of text decorations.

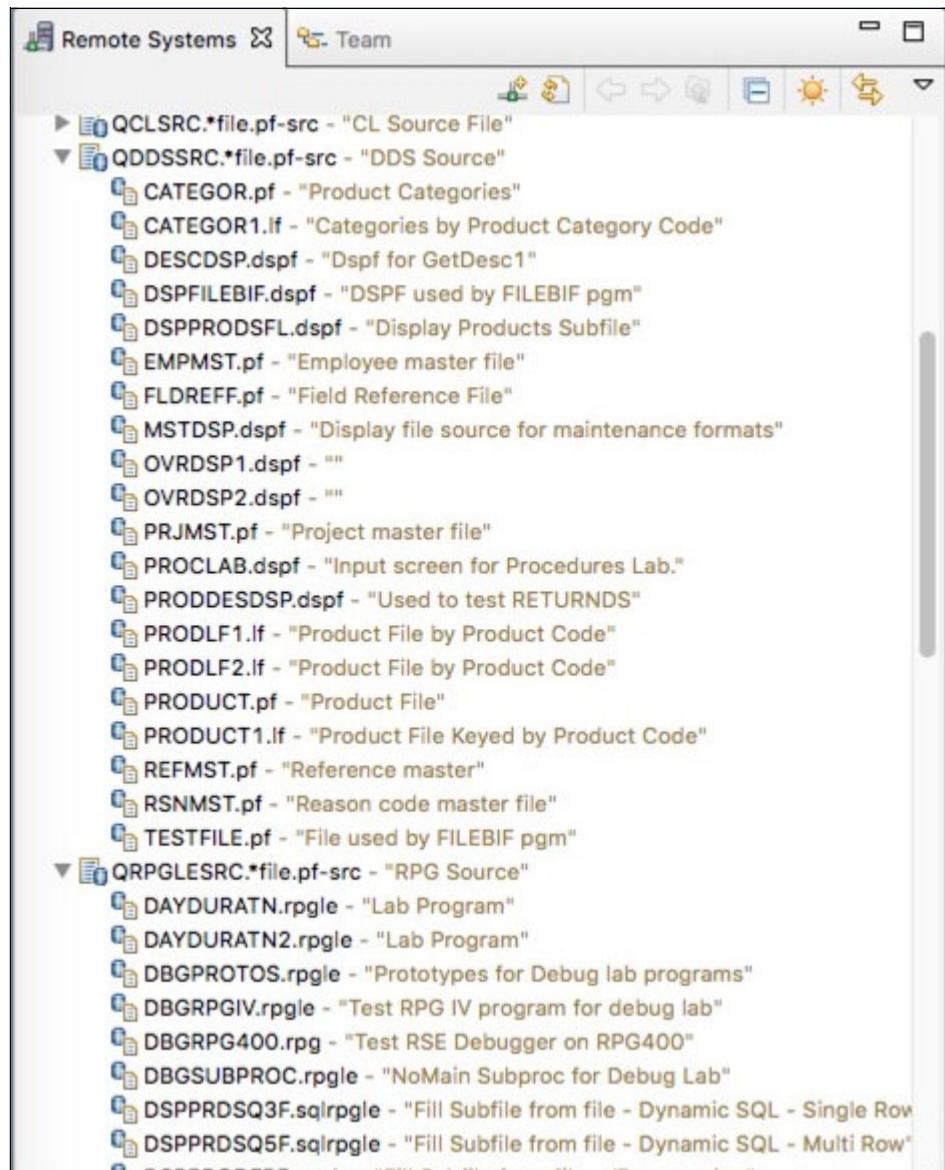


Figure 9-12 iSphere text decorations

- ▶ Working with data areas and user spaces

iSphere has great support for data areas and user spaces. You view and edit the content of them and design your own editor to show the structure of the data to make viewing and editing the content easier. There is also a tool to display the contents of a data queue.

- ▶ Other iSphere features

Other features of iSphere include a binding directory editor, a spooled file subsystem that supports text, HTML and PDF output, and support for creating RDi tasks from special tags in comments inside source code, and help with export and importing RSE filters. It also includes a compile pre-processor that retrieves the appropriate compile command and parameters to create an object from inside the source member for that object.

There are many more features in iSphere and more features are added frequently. For more information about some of the iSphere facilities, including how to download and install iSphere, see the following articles:

- ▶ *A Closer Look at iSphere*:

<http://www.ibmsystemsmag.com/ibmi/developer/rpg/isphere-details/>

- ▶ *iSphere Plug-in Expands RSE/RDi Toolset*:

<http://www.itjungle.com/fhg/fhg061615-story01.html>

- ▶ *More iSphere Goodies*:

<http://www.itjungle.com/fhg/fhg092915-story03.html>

- ▶ *Looking For Stuff With iSphere*

<http://www.itjungle.com/fhg/fhg070715-story01.html>

- ▶ iSphere Plug-in on SourceForge

<http://isphere.sourceforge.net/>

9.3 What is new in Rational Developer for IBM i

If you have been using RDi for a while, or if you tried some earlier version of RDi or its predecessors, you might not be aware of some of the new features that were added recently. This section is not a complete list of enhancements, but some of the more significant and helpful updates are included here.

The following RDi topics are covered in this section:

- ▶ 9.3.1, “Code coverage monitor” on page 352
- ▶ 9.3.2, “Integrated emulator” on page 354
- ▶ 9.3.3, “IBM i Access Client Solutions and Run SQL scripts” on page 355
- ▶ 9.3.4, “RPG formatter” on page 357
- ▶ 9.3.5, “More flexible outline view” on page 358
- ▶ 9.3.6, “Native Mac OS X support” on page 358
- ▶ 9.3.7, “Commenting and uncommenting in RPG, CL, and DDS” on page 359
- ▶ 9.3.8, “Hyperlink navigation within source member” on page 359
- ▶ 9.3.9, “Importing and exporting RDi configurations and push-to-client support” on page 359
- ▶ 9.3.10, “Smaller usability enhancements” on page 360

9.3.1 Code coverage monitor

Code coverage is a term that is used to describe a measurement of the degree to which the source code of a program is tested. The idea is that you start the code coverage monitor and then run your tests. Afterward, you can see various reports that show you how many lines of code and even which lines were run during the testing. This gives you an idea of what further test scenarios should be run to ensure that all the program functions are tested.

Code Coverage first appeared in RDi Version 9.1 and has been enhanced in several subsequent releases. Originally, Code Coverage worked only with programs that ran in batch. However, support for batch and interactive is now available as well as monitoring triggered using service entry points.

Code coverage uses the debugger. To use code coverage, you must first compile the program to be tested with the source debug option of **DBGVIEW(*ALL)**, **DBGVIEW(*SOURCE)**, or **DBGVIEW(*LIST)**, if it does not already have one of those options specified.

Compiler optimization can affect the report of the lines that are covered. For the most accurate results, compile with **OPTIMIZE(*NONE)**, which is the default for a system that is shipped from IBM.

Note: f portions of the application to be tested do not have debug information, only those portions of the application with debug information are included in the code coverage report.

When you run the RDi code coverage support, you get several options returned. Figure 9-13 shows the consolidated code coverage summary for a full program.

Code Coverage Summary			
Code coverage report (analyzed at Jan 26, 2016 1:31:12 PM, generated at Jan 26, 2016 1:31:12 PM)			
Element	Coverage	Covered	Total
*PGM RSELAB01/PAYROLLFFF	51%	123	243
PAYROLLFFF	51%	123	243
PAYROLLFFF.RPGL	51%	123	243
PAYROLLFFF0	57%	4	7
MAIN0	79%	22	28
REASONMAINTENANCE0	0%	0	45
EMPLOYEEMAINTENANCE0	87%	40	46
PROJECTMAINTENANCE0	0%	0	51
ISADDNEWRECORDREQUEST0	100%	3	3
ISADDPREVIOUSLYDELETEDRECORDRE	100%	3	3
VALIDATEFILETOMAINTAINSELECTION0	77%	10	13
VALIDATEACTIONCODE0	87%	13	15
VALIDATEADD0	89%	8	9
VALIDATECHANGE0	89%	8	9
VALIDATEDELETE0	80%	8	10
DISPLAYERROR0	100%	4	4

Figure 9-13 Code coverage summary report

In addition to the ability to see an overall report, you can actually drill down to see the individual lines that have been covered and which lines were not executed by the test case.

Figure 9-14 on page 354 shows the report that allows you to see each individual line.

```

// Display employee maintenance format
//
Exfmt EMPMNT;
If *INKC; //F3 = exit program
  Return;
Elseif *INKD; //F4 = return to main screen
  Return;
Elseif *INKE; //F5 = return to employee maintenance screen
  Iter;
EndIf;

// Determine update mode and perform record add or update
//
Select;
When isAddNewRecordRequest();
  ACREC = 'A';
  Write RCEMP;
When isAddPreviouslyDeletedRecordRequest();
  ACREC = 'A';
  Update RCEMP;
// Mark record deleted
When ACODE = 'D';
  ACREC = 'D';
  Update RCEMP;

```

Figure 9-14 Code coverage with individual lines shown

For more information about using the RDi code coverage monitor, refer to the *Running code coverage* topic in the IBM Rational Developer for i Knowledge Center:

http://www.ibm.com/support/knowledgecenter/SSAE4W_9.5.0/com.ibm.debug.pdt.codecovrage.i.doc/topics/tcc_run_about.html?cp=SSAE4W_9.5.0

There is also a hands-on lab available from IBM DeveloperWorks on code coverage. The following web page describes this code coverage hands-on lab:

https://www.ibm.com/developerworks/community/blogs/49773f8f-a20d-4816-86f2-44a2d862dbc1/entry/New_Code_Coverage_hands_on_lab_available?lang=en

9.3.2 Integrated emulator

Beginning with RDi Version 9.5, an integrated emulator session is available. To use it, right-click the **Objects** subsystem in the Remote Systems view and select **Host Connection Emulator**.

An emulation session opens in the editor area of your RDi workbench, as shown in Figure 9-15 on page 355. This emulation session is a separate job that does not interact directly with your Remote Systems connection.

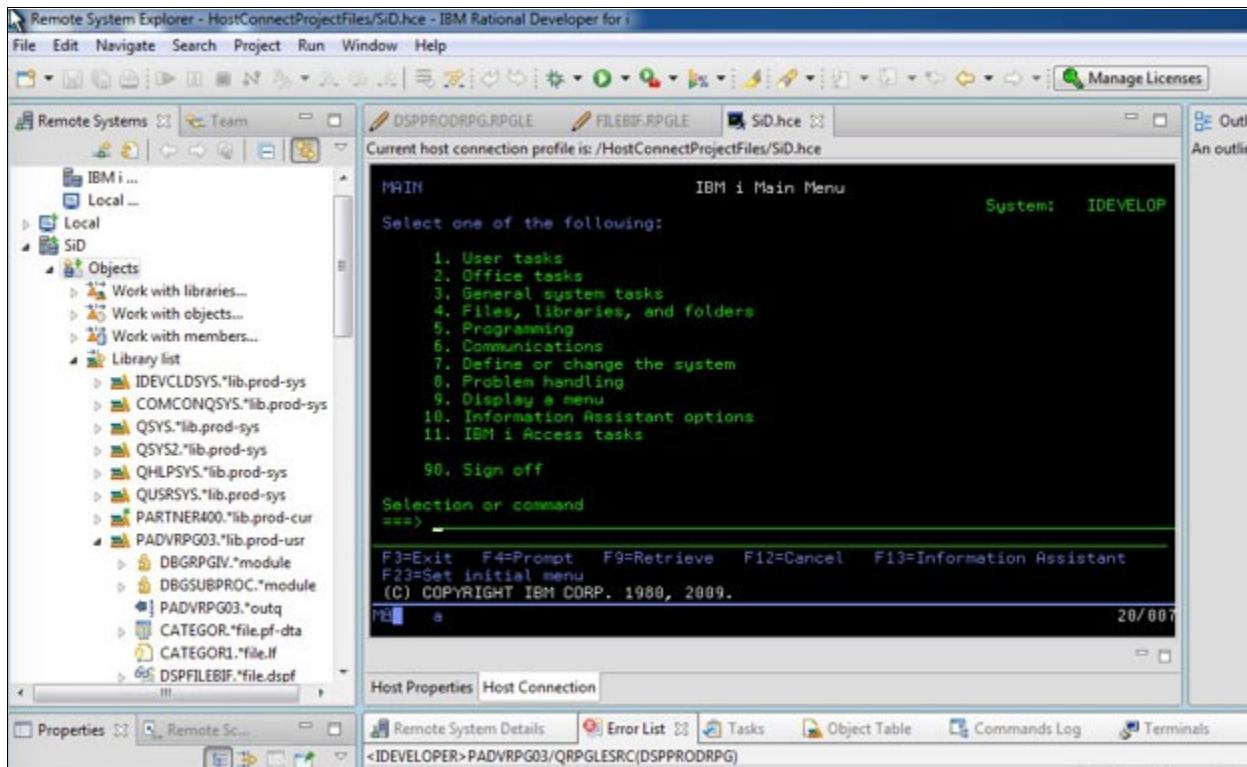


Figure 9-15 Integrated emulator

9.3.3 IBM i Access Client Solutions and Run SQL scripts

IBM i Access Client Solutions (ACS), including Run SQL scripts and its Visual Explain support, can now be started from RDi. This provides another way to start an emulation session. This emulation session runs in a separate window outside your RDi workbench.

In addition, you can start a Run SQL Scripts session, which can be helpful for testing SQL statements that you want to embed into your RPG code. Because the powerful Visual Explain support is also part of Run SQL Scripts, you have an easy way to analyze the SQL optimizer's plan for running the SQL statement, and index advice from the optimizer.

Figure 9-16 on page 356 shows how to highlight some SQL within RDi, click the **Source** tab and select **Launch Run SQL Scripts**.

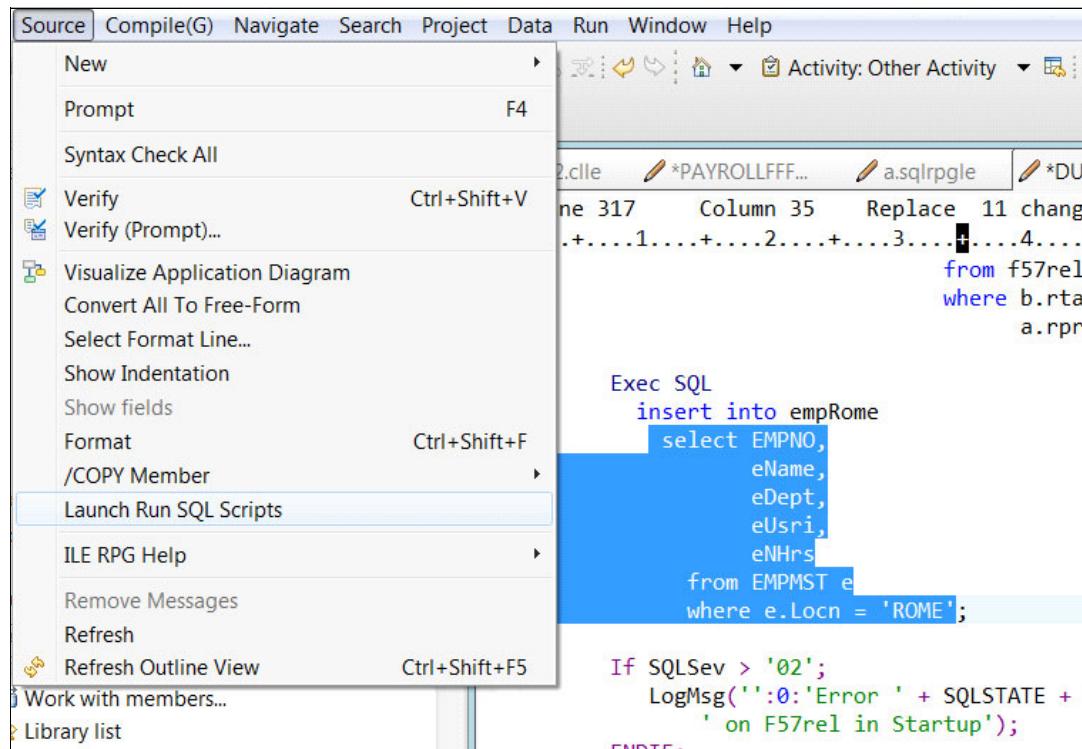


Figure 9-16 RDi integration with Run SQL scripts

The Run SQL Scripts window opens where you can run the selected SQL. You will have to update the source manually to replace program variables with actual values.

Figure 9-17 shows the Run SQL Scripts interface. With the latest support in RDi 9.5.1, there is a new Visual Explain link, which launches the visual explain function.

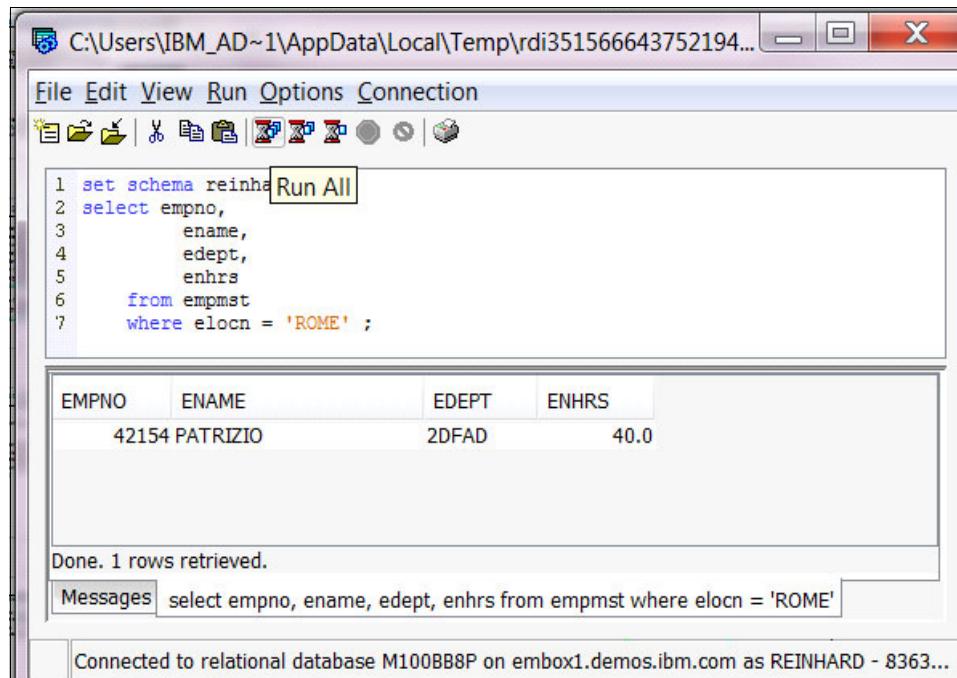


Figure 9-17 Run SQL Scripts window

9.3.4 RPG formatter

The ability to format, or more accurately, to re-format, free-form RPG code support was added in RDi Version 9.5.

For example, you might have a FOR or D0x loop that has a SELECT/WHEN block nested inside it and an IF/ELSE block nested inside that. Now, you must add another level of nesting with another IF/ELSE block that falls between the FOR/D0x and the SELECT block. One of the benefits of free-form logic is the ability to indent code to show the levels of nesting, which makes it easier for the next programmer to follow the logic. Therefore, it is *critical* that the indentation is *accurate*.

Even if the original code was indented correctly, introducing a new level in the middle requires shifting blocks of code to get the indentation to reflect correctly the new nesting levels. There are various ways to do this task manually in RDi, but they can be error-prone and time-consuming.

You can use the support that is now built into RDi to select a block of code and use the format option to indent the entire block appropriately based on the nesting levels of logic. You can use preferences to set up how many spaces to indent each level. The formatter supports all the free-format RPG statements, both logic and declarations.

You can use the built-in formatter for RDi on a specific block of code, or if no block is selected, the entire member is formatted. Pressing Crtl+Shift+F starts the formatter, or you can start it by using the Source menu by right-clicking and selecting **Source**.

This formatting support does not convert fixed-format RPG to free format. It re-formats only code that is coded in free format. The Convert All to Free-form feature in the Source menu in RDi deals only with logic (no declarations or H spec code), and it is a limited free-format conversion in many cases. Any logic statement that it cannot handle is surrounded by /End-Free and /Free directives. Therefore, its usefulness is limited.

There are some good options that plug into RDi that do a better job at converting fixed format to free-format RPG, including logic and declarations.

The RPG Toolbox from Linoma Software has an option that can convert either an entire member or a block of selected code to free format. It also has a re-format option that is similar to the one that is built into RDi, but it has some slightly different options for the reformat behavior.

Likewise, Arcad Software also offers ARCAD-Transformer, which converts entire source members from fixed-format RPG to free format and plugs it into RDi. Figure 9-18 on page 358 shows the before and after of some code that was re-formatted using the formatter function. You can launch this support by clicking the CTRL-Shift F keys. The formatting support can also be customized to allow you to specify how much you want to indent after certain code constructs.

```

Line 12      Column 37      Insert 34 c
.....+....1....+....2....+....3....+....4
      DCL-DS *N;
      DCL-SUBF select CHAR(10);
      name CHAR(10);
      DCL-SUBF address CHAR(25);
      END-DS;
      if x=5;
      y=3;
      elseif x =4;
      y
      =
      2;
      endif;

Line 9       Column 26      Insert 34 chan
.....+....1....+....2....+....3....+....4...
      DCL-DS *N;
      DCL-SUBF select CHAR(10);
      name CHAR(10);
      DCL-SUBF address CHAR(25);
      END-DS;
      if x=5;
      y=3;
      elseif x =4;
      y
      =
      2;
      endif;
  
```

Figure 9-18 Before and after formatting from fixed-format to free-form format

9.3.5 More flexible outline view

The RPG outline view is a favorite feature for many RPG developers. It has been improved over the years in several ways. The Outline view has been dynamically updated with changes made to the source code and new features added, including the ability to filter the outline view.

The filter box near the top of the input in some text, for example, a full or partial name of a variable or routine, and the outline view then filters everything except items containing that text. This feature is helpful for finding items quickly in an outline view for a large program or one with many definitions.

Unreferenced items can now be hidden from the outline. This is helpful when the code copies in many items that a particular program does not need, such as copying many unreferenced prototypes into a source member.

9.3.6 Native Mac OS X support

Until recently, RDi was supported only on a Windows platform and some specific Linux platforms. RPG developers who used the Mac OS X were forced to run a virtual machine or dual-boot environment in Windows to use RDi. Now, RDi is supported natively on OS X Version 10.11 (El Capitan) or later.

There are some limitations in the support for OS X and some differences in behavior, at least in the initial release of the OS X support at the time of writing (Version 9.5.1). Most notably, syntax checking and the program verifier is not supported at the time of writing. Some

keyboard shortcuts might be different from the ones that are used in Windows. Bi-directional text is not currently supported and the integrated 5250 emulator is not supported, although the ACS emulator is supported through the Remote Systems view.

Even with the current limitations, the ability to run RDi without requiring a Windows virtual machine is advantage to those IBM i developers who use Mac workstations.

9.3.7 Commenting and uncommenting in RPG, CL, and DDS

The ability to comment (or un-comment) a single line of code or a selected block of code has been available for a while. More recently the same capability has been added to the CL and DDS editors.

Commenting can be done by using the shortcut or **Ctrl+/-** or by selecting **Source → Comment** from the context menu. Un-commenting is done with **Ctrl+**. Both can either be applied to a single line (where the cursor is positioned) or to a selected block of code.

9.3.8 Hyperlink navigation within source member

Hovering over a variable in an RPG source member shows the definition of it. To navigate to the definition inside the source member, hold down the **Ctrl** key and the variable turns into an underlined hyperlink that when clicked will take you to the definition. The **Alt-Left** keyboard shortcut (or use the left arrow in the toolbar) takes you back to the original reference to the variable.

Continued use of the **Alt-Left** and **Alt-Right** keyboard shortcuts can navigate all the way back through all your edits to the file.

This is a similar kind of navigation to the subroutine or internal subprocedure link that happened by using **F3** when positioned on the routine name. The **Alt-Left** shortcut also returns you to the original reference to the routine. In earlier releases, the **Alt-Q** shortcut was used to return to the original position. This was changed in more recent versions to be more consistent with the extended use of the **Alt-Left** and **Alt-Right** navigation through various edits.

9.3.9 Importing and exporting RDi configurations and push-to-client support

To help with administration of RDi and sharing of RDi customizations, the ability to export and import configuration information is supported. This can be useful for an individual developer to be able to back up and restore his specific configuration information or to share some of the customization done with other developers.

The push-to-client support enables automatic distribution of RDi configuration information to many developers. Some items that can be distributed this way includes preferences, filters, connection definitions, templates, and snippets.

A video demonstrating some scenarios for this support can be found here:

<https://www.youtube.com/watch?v=jEBKyI8QjT>

9.3.10 Smaller usability enhancements

Many smaller, but very helpful, usability enhancements have been made to existing functionality in RDi recently. This includes the following functions:

- ▶ F1 help support for RPG built-in functions (BIFs).
- ▶ Preference to customize the way the Alt-S split line works.
- ▶ Member filters can now filter based on member text.
- ▶ The show block nesting support that helps to decipher deeply nested logic blocks now includes an indication of the level of ELSE and ELSEIF statements as well as a preference for the color of the lines and arrows.
- ▶ Kerberos authentication for IBM i connections works well with single sign-on environments.

9.4 Using the RDi debugger

The biggest challenge when debugging RPG programs with RDi is getting the debug session started. There are two primary ways to begin a debug session with RDi:

- ▶ Using a service entry point (SEP).
- ▶ Creating and using a debug configuration.

9.4.1 Starting a debug session using service entry point

For most situations, using a service entry point is the easiest method to start a debug session.

To set a service entry point, find the program (or service program) that you want to debug in the Remote Systems view. Right-click on the program or service program and select **Debug or Code Coverage (Service Entry)** → **Set Service Entry Point**.

It can take several seconds for anything to happen, but eventually you see a confirmation message and the Service Entry Points view opens in the workbench. Your program name appears in the list of Service Entry Points. The default settings include *All modules in the program, *All procedures in the program, and your user ID. Those details can be changed from the Service Entry Point view.

Figure 9-19 shows the Services Entry Point view.

Library	Program	Program Type	Module	Procedure	User ID	Connection	Enabled
RSEEX25	DSPPRODFR2	*PGM	*ALL	*ALL	RSEEX25	I DEVELOPER.idevcloud.com	Yes

Figure 9-19 Service Entry Points view

Setting a service entry point starts a monitor on the host. Whenever the specified program is started by the specified user profile (by default, this is your user profile) anywhere on your IBM i host system, the host debugger puts the job where the program is running into debug mode and then notifies RDi on your workstation.

It does not matter whether the program runs in a 5250 emulator, a batch job, or a server job. The system is monitoring for any job that begins to run the specified program under the specified user profile. When that happens, the following steps occurs:

1. The Debug perspective in RDi opens (if it was not already open).
2. The specified program is put into debug mode.
3. The source for the program is opened in the editor (if it was not already open).
4. The debugger waits before the first executable statement for your interaction.

Invoke the SEP program using the specified user profile from any job in any environment to cause all of those actions to occur. It might mean calling the program from an interactive job or submitting a batch job or even invoking the program using a web server or database server job.

Your debug session is started and is waiting for you to tell it how to proceed. You might want to set some breakpoints or step through the code, which are described in 9.4.4, “RDi debug activities” on page 362.

Note: You do not need to do anything special in the job to be debugged. The system detects that the requested program has been started by the appropriate user profile and puts that job into debug mode automatically for you.

9.4.2 Starting a debug session using a debug configuration

There is a second way to begin a debug session, that is, by creating and using a debug configuration. Service entry points are not enabled for all types of programs. Only programs that are created with an ILE compiler (for example, programs that are created from member type RPGLE or CLLE) can use a service entry point to begin a debug session.

However, that does not mean that non-ILE language programs (for example, RPG or CLP) cannot be debugged in a session that was started with a service entry point. It simply means that a non-ILE program cannot be the program that starts the debug session. One solution in some cases might be to create a simple RPGLE or CLLE program that calls the non-ILE-language program so that you can set a service entry point on it and then step into the non-ILE program. However, sometimes that might not be practical to do, in that case, you can still debug those programs by using a debug configuration.

One other situation where you cannot use service entry points is when the program is already running, such as in a never-ending job. A debug configuration works for those situations.

There are multiple ways to create a debug configuration and multiple types of configurations. The most useful type of debug configuration is typically a job configuration. It does not matter whether the job is interactive, batch, or server, it is still a job, and a job debug configuration works to debug a program in that job.

The simplest way to create a job debug configuration is to find the job to be debugged in the Jobs subsystem in the Remote Systems view in RDi. The Jobs subsystem by default is below the Objects, Commands, and IBM i Contexts subsystems, and above the IFS Files subsystem. There are many built-in filters to help find the job to be debugged, and you can also create your own job filters to make the search a little faster and easier.

Tip: If you are looking to debug a job with your own user profile, for example, you might find the built-in "My active jobs" filter useful.

After finding the appropriate job, right click on it and select **Debug (Prompt)** to start a dialog box where you can supply the qualified name of the program(s) (or service program) you want to debug.

If you need to use a different debug configuration type you can create any other type of IBM i debug configuration from the Debug Configurations option from the Run menu. There are six different IBM i specific configuration types available, including specific ones for batch, multi-threaded, and incoming remote debug sessions.

9.4.3 Debug server and preferences

Regardless of which method you use to start the debug session, you might need to start the debug server or set some preferences to get your debug session going.

The debug server runs on the IBM i host system and needs to be running before you can debug. If it is not running, a notification message is displayed in RDi. The message offers an option to start the debug server. If you do not have sufficient authority to start the debug server on the host, you might need to have a system administrator start it using the Start Debug Server (**STRDBGSRV**) CL command. Ideally that command could be included as part of the system start up routine that automatically runs during the IPL process.

There is an IBM i Debug preference page in RDi. If you normally need to specify *Yes for Update Production Files when using the Start Debug (**STRDBG**) command, then you should be sure to select Update Production Files in the Preferences. It is possible to specify that feature when using a configuration to start your debug session, but not when using service entry points. It is advised that if you need to have the Update Production Files set to *Yes normally, go ahead and set that on your RDi IBM i Debug preference page.

9.4.4 RDi debug activities

Once you have successfully started your debug session, you can set break-points and step through the code line by line, just as you could with the host-based debugger. Line breakpoints can be set (or removed) using the context menu on an line of code in the editor when in debug mode or by double clicking in the far left margin, to the left of the source sequence numbers.

Watch breakpoints work the same in the RDi debugger as they do in the host-based debugger. The easiest way to set a watch breakpoint typically is to select the variable name in the source and select **Add Watch Breakpoint** from its context menu. Note that in the resulting dialog box, a value of 0 in the "Number of bytes to watch" entry field means to watch the entire variable value.

Similarly, conditional breakpoints can also be used. After setting a line breakpoint, right click on the breakpoint, either in the source view or in the Breakpoints view, and select **Edit Breakpoint** from the context menu. Click **Next** on the first dialog page to get to the ability to enter the conditional expression entry box on the second dialog page.

Stepping through the code can be done using the various step icons in the Debug Perspective toolbar or by using options under the Run menu or by using function keys which are shown next to the Run menu Step options. The Step and Step Into options are there and work as they did in the host-based debugger.

In addition, there are two additional types of steps available:

- ▶ Automated Step Into. The debugger continually repeats the step function while you can watch the logic flow and the values of the variables.
- ▶ Step Return. The debugger returns to the calling program or procedure following a Step Into.

You can look at and modify the values of variables while debugging. To look at a variable's value, you can hover over the variable name in the source view. In order to monitor the values of specific variables while stepping through the code or while going from one breakpoint to the next, the debugger Monitors view is helpful.

To add a variable to the Monitors view, select the variable name by selecting it in the editor and select **Monitor Expression** from the context menu of the selected variable. Alternatively, a variable can be added by name using the context menu or the plus sign in the Monitors view.

Figure 9-20 illustrates the use of some of the capabilities of viewing the value of a variable during debug. Note that any variable value that changed since the last time the logic was halted (a breakpoint or a step) turns red to make it easier to tell what actions happened recently to the variable's values. A variable's value can also be changed from the Monitors view by clicking on the value and a dialog box appears to allow entry of a new value.

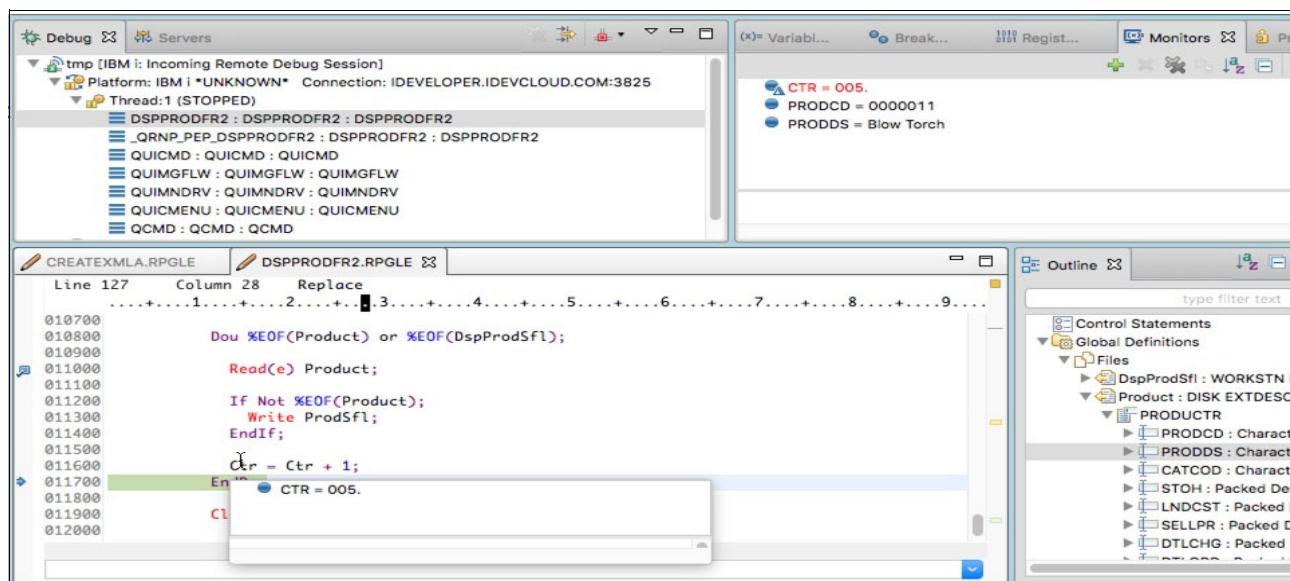


Figure 9-20 Viewing variable values in RDi debug

Note: You might notice that there is a Variables view in the Debug Perspective overlaying the same space where the Monitors view is. For small procedures or programs, this can be an effective alternative to view variable values because it contains all the variables in the source being debugged. However, since most RPG programs tend to be relatively large with large numbers of variables, that view sometimes slows down the debug process. Bringing the Monitors view to the foreground is useful not only because you can easily watch any changes to a few relevant variables while debugging, it also means the Variables view does not need to be refreshed at each step or breakpoint, which can improve the performance of the debug session.



Modern RPG comparison as viewed from a young developer

Transistioning between the old RPG fixed-format and the new modern RPG or free-format RPG can seem difficult for both for the experienced RPG programmer and the younger developer.

Kody Robinson is a younger developer. He is currently two years out of university and has been working as an RPG programmer these past two years. His company has been embracing modern RPG, the tools, and methodologies that have been discussed in this book. In the following sections of this chapter, he is going to break down some of his favorite code concepts and look at the difference between the old and new. This chapter is not intended to be a tutorial or reference manual to show all the possible differences, but rather look at modern RPG from the eyes of a young developer that has been making the change from old to new and some observations. There are some great pointers in this chapter for all developers.

The following topics are discussed in this chapter:

- ▶ 10.1, “Fixed format verses free-format” on page 366
- ▶ 10.2, “File-Specs” on page 367
- ▶ 10.3, “Tables” on page 368
- ▶ 10.4, “Parameters” on page 368
- ▶ 10.5, “Key lists” on page 369
- ▶ 10.6, “Array and error lookups” on page 370
- ▶ 10.7, “Calling programs from a program” on page 371
- ▶ 10.8, “Read commands” on page 372
- ▶ 10.9, “Evaluates, checks, and scans” on page 373
- ▶ 10.10, “Select statements” on page 374
- ▶ 10.11, “Substring and concatenate” on page 375
- ▶ 10.12, “Go-To commands” on page 376
- ▶ 10.13, “Embedded SQL in free-format 101” on page 377
- ▶ 10.14, “Built-in functions you need to know” on page 379

10.1 Fixed format verses free-format

Take a look at the code shown in Figure 10-1, this is a program that has been converted it to RPGLE (disclaimer, this is not real ILE). In addition to very outdated code, you are currently looking at a very outdated programming environment. What new developer coming out of college actually wants to look at something like this for 8 hours a day to program in? There is no ability to highlight, right click, or copy and paste.

```

CoTumns . . . : 6 100
SEU==>
FMT C CLONOIFactor1+++++OpCode&ExtFactor2+++++Resultc+++++Len++D+HiLoEq..
0040.00 CL1 APINVN SETLL ARSHSTL1
0041.00 CL1 READ ARSHSTL1 30
0042.00 CL1 *IN30 IFEQ *ON
0043.00 CL1 APINVN ORNE ARSIV#
0044.00 CL1 MOVE *ON *IN40
0045.00 CL1 ENDIF
0046.00 CL1 DOWEQ *OFF
0047.00 CL1 APINVN ANDEQ ARSIV#
0048.00 CL1 ADD ARSSAM L1SAM 11 2
0049.00 CL1 ADD ARSSAM L2SAM 11 2
0050.00 CL1 ADD ARSSAM LRSAM 11 2
0051.00 CL1 ADD 1 CNT 1 0
0052.00 CL1 EXCEPT PRTDTL
0053.00 CL1 READ ARSHSTL1 30
0054.00 CL1 ENDDO
0055.00 CL1 CNT IFGT 2
0056.00 CL1 EXCEPT PRTL1
0057.00 CL1 ENDIF
0058.00 CL1 CLEAR CNT
0059.00 CL1 CLEAR L1AP

```

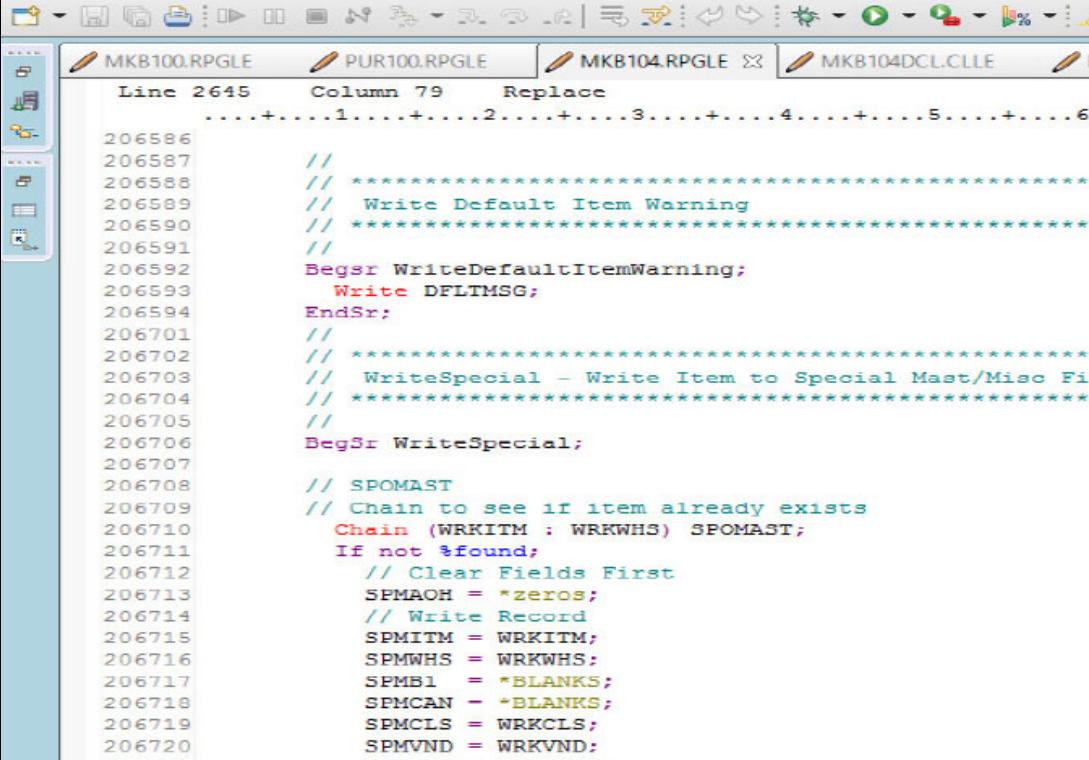
Figure 10-1 Fixed format RPG ILE code

Let's say you have a new developer and you want them to look at a problem in the program shown in Figure 10-1. Syntax is pretty easy to learn if you have any programming knowledge at all, so we are going to assume that they know that SETLL is an operation code (line 0040.00).

Look one line down at line 0041.00. Odds are is that they are going to ask you what the random number means to the far right. It's simple right? I mean who wouldn't know that the number 30, in this particular instance, means if a record is found or not in the logical file ARSHSTL1. Doesn't every company use the number 30 for the same thing? Isn't that kind of like an unspoken rule of programming? No, no it's not. Jumping down a few lines more you see *IN30. So now you not only have one random number in your code which doesn't give you any insight into what it does or is used for. I hope our young developer hasn't ran off yet.

Let's make a comparison between Figure 10-1 and the one shown in Figure 10-2 on page 367. Notice anything different?

- ▶ Notice how there are actually more colors than green and black?
- ▶ Notice how the *INXX (indicator) is no longer there?
- ▶ Notice how you can actually use more than 6-10 characters naming a field?
- ▶ Notice how this code looks exactly like every other modern language on the planet?



The screenshot shows a window titled 'MKB100.RPGLE' with tabs for PUR100.RPGLE, MKB104.RPGLE, and MKB104DCL.CLL. The code is in fixed-form, with lines numbered from 206586 to 206720. The code is a series of comments and declarations related to item processing, including sections for 'Write Default Item Warning', 'WriteSpecial', and 'SPOMAST'. The code uses various RPG keywords like //, *zeros, WRKITM, WRKWHS, etc.

```

Line 206586      Column 79      Replace
.....+.....1.....+.....2.....+.....3.....+.....4.....+.....5.....+.....6
206586
206587      //
206588      // ****
206589      // Write Default Item Warning
206590      // ****
206591      //
206592      Begsr WriteDefaultItemWarning;
206593      Write DFLTMSG;
206594      EndSr;
206701      //
206702      // ****
206703      // WriteSpecial - Write Item to Special Mast/Misc Fi
206704      // ****
206705      //
206706      Begsr WriteSpecial;
206707      //
206708      // SPOMAST
206709      // Chain to see if item already exists
206710      Chain (WRKITM : WRKWHS) SPOMAST;
206711      If not %found;
206712          // Clear Fields First
206713          SPMAOH = *zeros;
206714          // Write Record
206715          SPMITM = WRKITM;
206716          SPMWHS = WRKWHS;
206717          SPMB1 = *BLANKS;
206718          SPMCAN = *BLANKS;
206719          SPMCLS = WRKCLS;
206720          SPMVND = WRKVND;

```

Figure 10-2 Same RPG ILE code transformed to free-format

The rest of this chapter does a few comparisons between some fixed-form RPG and its counterpart, free-form. Most of this is basic knowledge, but this should help developers who aren't quite ready to make the switch to modern RPG.

10.2 File-Specs

Looking at Figure 10-3, you can see a few differences between fixed-form file specs and free-form file specs. Probably the most noticeable difference is being able to actually tell the compiler what you want to be done with your files (for example, update, delete, or output). The compiler also assumes it is disk and external, so keying those aren't necessary. Take a look at INVMSTR for a moment. Notice how in the free-form example all you have to put is "keyed?" That's because the compiler assumes that the usage is input if no usage is declared.

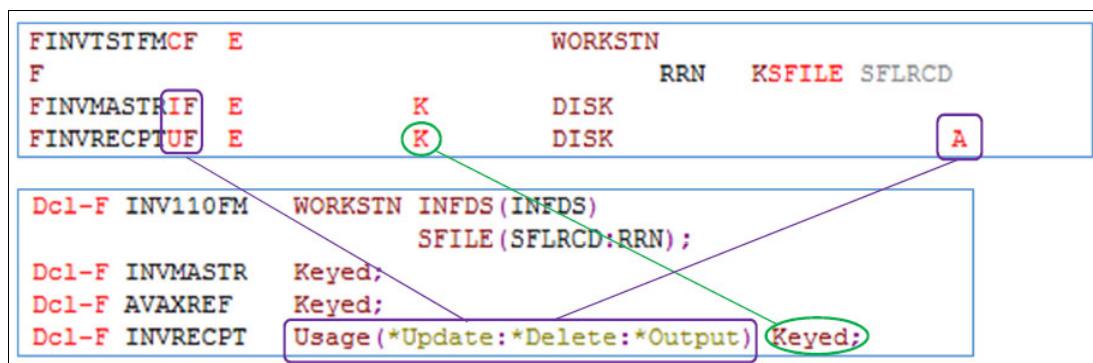


Figure 10-3 File specs comparison

10.3 Tables

Take a look at the top portion of Figure 10-4. If you have never seen RPG code before, you would think that someone threw some letters and numbers at a wall and the turnout is what is pictured. This example is shown using RD_i, so you can at least see some separation with the colors. Even though these are included in a table, they are still standalone variables, the bottom portion of Figure 10-4 uses the Dcl-S code. This follows the same structure that all new RPG free format follows. Declare the name, specify the type and length (Char(2)), name how many outputs are in the table, and so on.

Figure 10-4 Tables comparison

10.4 Parameters

Parameters are important. That is how you pass information back and forth between program, CLs, and a lot of other things. The old way of passing parameters is pretty easy. Just make a PLIST and add some values. Well, it may be easy, but it is not that descriptive and not very efficient. Why? Well for one you are still limited to your field lengths. That is a pretty obvious one. Another one is there is no way to exactly tell which program those parameters are coming from. Do these parameters come in from just one program? Or are these parameters a part of a bigger array of company programs? With free-format parameters, you can actually say which parameters deal with what programs or CLs.

Another great comparison (and one of the reasons of why you should use free-format) is you have much more descriptive declarations at your disposal. You state that the parameter is character, decimal, and so on. Figure 10-5 on page 369 shows how you can define “like” in a much easier, more descriptive way.

You might either think that MKB104 is either a very structured program name or one that is not descriptive at all. Either way you look at it, you're right. If you are like us and name program like the ones pictured (I mean, you really can't do too much with limited program name lengths) then you probably want to use this free-form style anyways. Not everyone (especially a developer) is going to know what MKB104 does. There's a fix to that, and the fix is to use free-format. Then Pgm_MKB104 can be called descriptive names like "SpecialOrderEntry," "PrintAdjustedSales," "EndOfDay," or any other descriptive thing you want to say. This is just another example of how much more effective and productive you can be with free-format.

*ENTRY	PLIST		
	PARM	MBMORD	10
	PARM	WRKWHS	2
	PARM	LSEQ	1 0
	PARM	SCROCD	1
// Prototype for MKB100			
Dcl-Pr Pgm_MKB104 ExtPgm('MKB104');			
MBMORD Char(10);			
WRKWHS Char(2);			
LSEQ like(LSEQ#);			
SCROCD Char(1);			
 End-Pr;			

Figure 10-5 Comparison of parameters

10.5 Key lists

Now onto one of my favorite things about RPG free-format, and that's chaining. Back in my RPGIII days I would hate having multiple files to work with because of the different key lists you would have to have just to get the data from the files. Let's face it, hardly any programs just deal with one or two files. Here is where RPG free-format comes in and completely gets rid of the need for key lists.

Previously you would have to define a distinctive KLIST and put the fields that make up that key. In free-format, just follow the Chain command with the fields you want (shown in Figure 10-6 on page 370 as PRMPO# and PRMRLS) and then the file that you want to chain to. It's that simple. No more extra code to confuse a new developer even more. It is just the code that you need in order to get the data that you want.

While we are talking about chaining, it might be a good idea to talk about indicators and how they relate to the command. You used to be forced to have an indicator to tell whether or not your data was found. It is still just as important, just not the correct way to go about it. In Figure 10-6 on page 370 you now just use an IF indicator to condition your chaining. You can either condition it with "If %found" or "If not %found." So depending on your programming style, you can condition if you want your logic to follow if there is data out there matching your keys, or if there isn't, the choice is yours.

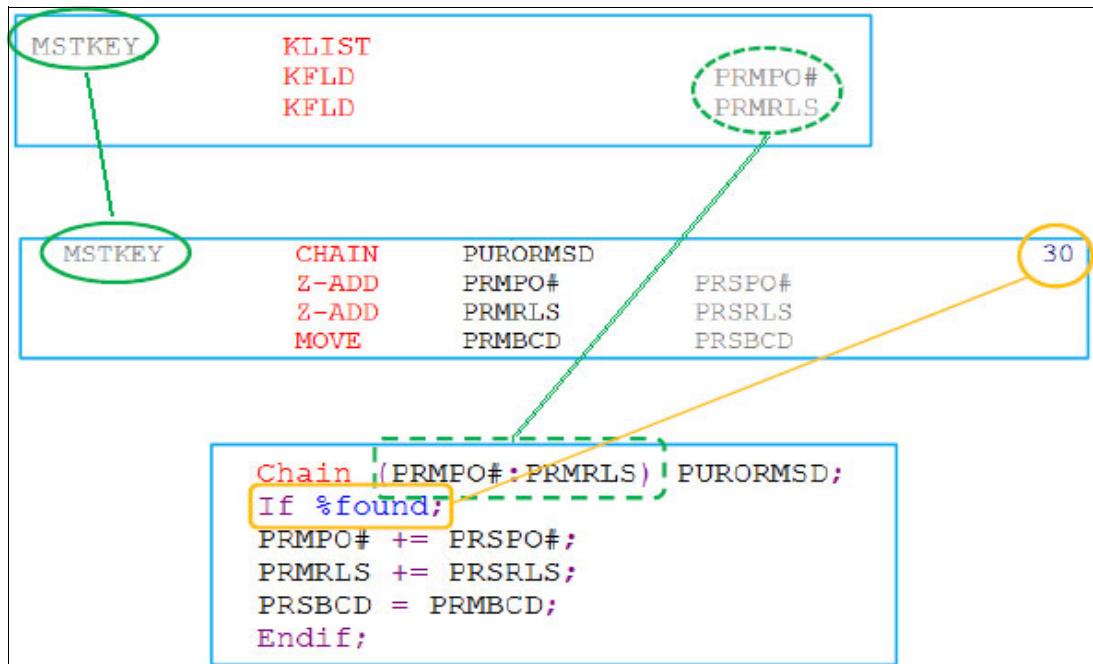


Figure 10-6 Comparison of key lists

10.6 Array and error lookups

Probably one of the more common things to do in a RPG program is to have some condition happen and then look up the appropriate response in an array. There are several ways to accomplish this, Figure 10-7 on page 371 shows the standard that is followed. You set some conditions, then if those conditions are met, you turn on some indicators, then move a unit from an array onto the screen, then output it in an updated subfile. Doing it the old way still has a few setbacks like fixed field lengths, use of non-descriptive indicators, as well as a few other small things. In the free-format style however, you have a much better understanding of what is going on, especially if you are a new programmer looking at this code for the first time.

Instead of using the operation code SETON, you just “turn on” the indicators (if you choose to still use traditional indicators) like shown in the bottom portion of Figure 10-7 on page 371. You can either use *INXX = *On or you can do it like the one pictured in Figure 10-7 on page 371 and use the traditional “1” for on and “0” for off. You now have the %TLookup code that pretty much functions just like the traditional operation code, so no special learning required.

In the example, it is setting *in89 on if it is given an error from the array. From here on out, it functions the same. The “08” is the spot in the array that we want to pull from, then we are moving it all the way to TABOUT, which will be moved to WERR. The Update SFLRCD is written the exact same, as well as the leave and else.

Suspended Item				
MIMRCD	IFEQ	'S'		
MBMSOS	ANDEQ	0		
	SETON			9098
'08'	LOOKUP	TABERR	TABOUT	89
	MOVE	TABOUT	WERR	
	WRITE	DSPERR		
	MOVE	WRKITM	HLDITM	21
	MOVE	WHS	HLDWHS	2
	UPDATE	SFLRCD		
	LEAVE			
	ELSE			
<pre>// Suspended Item If MIMRCD = 'S' and MBMSOS = 0; *IN90 = '1'; *IN98 = '1'; *IN89 = %TLookup('08' : TABERR : TABOUT); WERR = TABOUT; Write DSPERR; HLDITM = WRKITM; HLDWHS = WHS; Update SFLRCD; Leave; Else;</pre>				

Figure 10-7 Comparison of array and error lookups

10.7 Calling programs from a program

Sometimes in RPG free-format you have to put in a little more work to make the overall program flow better and be more descriptive. Internally calling programs (and subprocedures) fall into this category. When calling programs in the older days, you would just use the call command, follow with the program in quotes, and then pass in any parameters that you might need. Pretty simple, right? Yes, but once again you are confined to a number of limitations.

It is still pretty simple to do this in free-format, and often a lot better. As discussed in 10.4, “Parameters” on page 368, the same rules apply here. This program shown in Figure 10-8 on page 372 passes in parameters MBMORD, WHS, and PKSEQ into MKB142CL to do some processing. At the top of the program, a prototype is defined just like before. The naming standard is Pgm_XXXXX, however you can have that as descriptive as you want as long as you define the ExtPgm as your actual program name. Then underneath, you define your parameters and their attributes.

Figure 10-8 on page 372 shows calling a program in both fixed-form and free-format. When calling the program, it is pretty much the exact same as in fixed-format, only it looks a little more modern. The call command is no longer necessary. Now just code in the Pgm_XXXXX (or whatever you coded it to be) and follow it by your parameters within parentheses, separated by colons if more than one are present.

```

Qty Packed > Qty on Hand - send e-mail
  MBMSOS      IFEQ      0
  EML          ANDEQ    'Y'
  Z-ADD        1          PKSEQ
  CALL         'MKB142CL'
  PARM
  PARM
  PARM
  ENDIF

// Prototype for MKB142CL
Dcl-Pr Pgm_MKB142CL ExtPgm('MKB142CL');
  MBMORD      Char(10);
  WHS         Char(2);
  PKSEQ       Packed(2:0);
End-Pr;

// Qty Packed > Qty on Hand - send e-mail
If MBMSOS = 0
  and EML = 'Y';
  PKSEQ = 1;
  Pgm_MKB142CL(MBMORD : WHS : PKSEQ);
EndIf;

```

Figure 10-8 Comparison of calling a program

10.8 Read commands

Read commands are one of the more commonly used commands, since you really cannot do too much with files without them. Not too much is different here. As a developer, you can still use READ, READE, READP, and so on. The only difference is instead of leading with factor 1, you start the line with your READ, then your field(s), then your file. Figure 10-9 on page 373 shows a couple of examples of using READ in both old and new styled RPG.

```

C* Load up to 8 records into subfile page
C          DO      8           RCVRRN
C* Load existing records
C     MBMORD      READE      RECMBD
C*
C* Load existing DB records
C     *IN31      IFEQ      *OFF
// Load up to 8 records into subfile page
For RCVRRN = 1 To 8;
// Load existing records
  ReadE MBMORD RECMBD;
  *IN31 = %Eof;

C* Not authorized to order from another whs
C     MBMORD      SETLL      MKBORDTL
C     MBMORD      READE (N)  MKBORDTL
C     *IN31      IFEQ      *OFF
C     MBDWHS      IFNE      'AL'
// Not authorized to order from another whs
  Setll MBMORD MKBORDTL;
  ReadE (N) MBMORD MKBORDTL;
  *IN31 = %Eof;

```

Figure 10-9 Comparing read commands

10.9 Evaluates, checks, and scans

Evaluates are probably the easiest commands to move from fixed-format to free-format because all you do is take away the command itself. In fixed-form, you would have **EVAL THISFIELD = SomeCalculuationsOverHere**. In free-form, all you do is take out the **EVAL** and let the code say **THISFIELD = SomeCalcualtionsOverHere**. Check the EVALs circled in orange in Figure 10-10 on page 374.

Much like the %TLookup command, the CHECK command now functions about the same way, meaning you condition it with a "%." Instead of leading with the %Check command (green circle in Figure 10-10 on page 374), you lead with the result field. In this case, it is the NUM field. Essentially it is the same process, except for your factors swap around. Your result field goes in the front, followed by the %Check command, then conditioned within parentheses are your factor 1 and factor 2.

The same works for \$Scan, too. In Figure 10-10 on page 374 circled in blue, you lead with your result field, followed by the command. Then condition your factor 1 and 2 within parentheses, compile, and you have a working %Scan command.

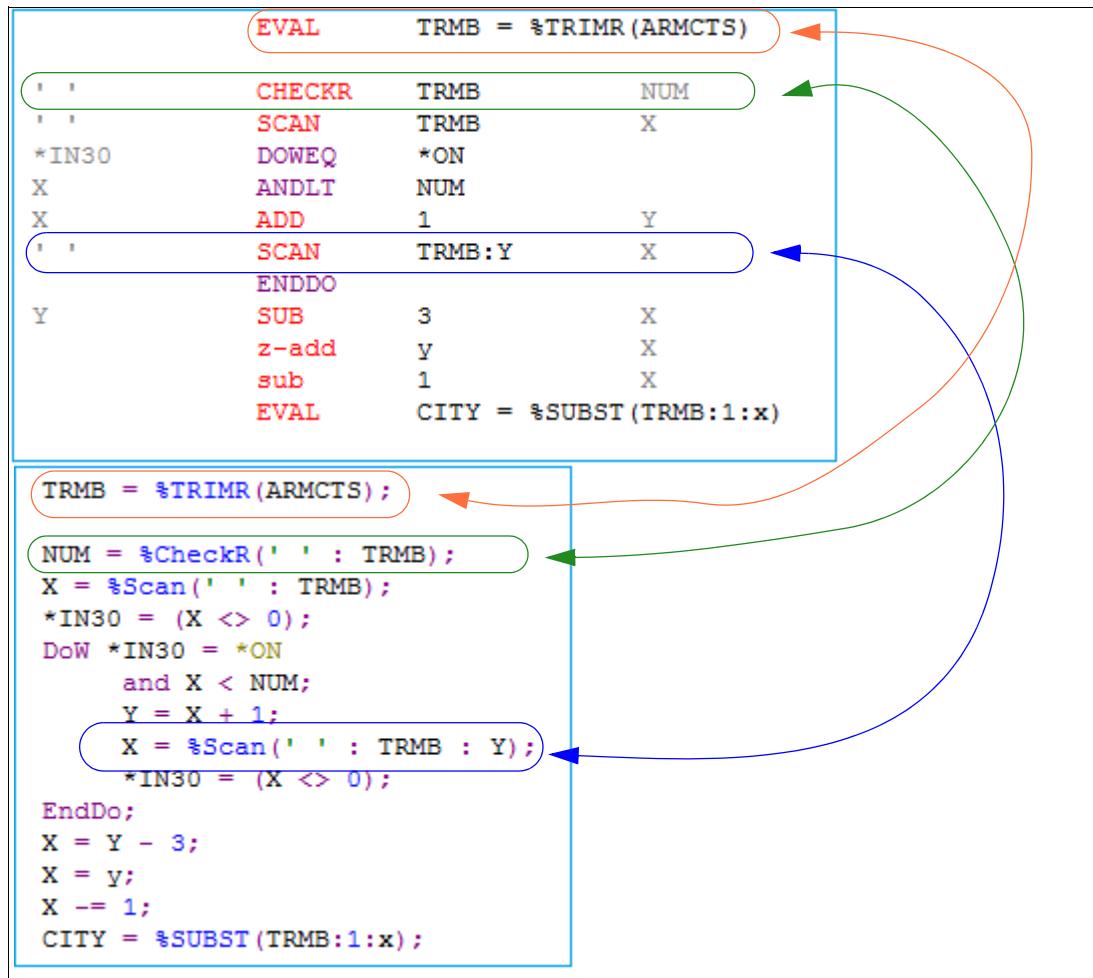


Figure 10-10 Compare of evaluates, checks, and sums

10.10 Select statements

Selects are a great way to condition things, especially if you do not want to use a lot of IF/ELSE combinations. In the fixed-format way, you start your select with the command, then use your when clauses (WHENEQ, WHENNE, and so on) to condition some calculation, output, or call command. In free-format, the only thing that is different is that you use operators instead of words (for example WHENNE = "<>").

In Figure 10-11 on page 375, the user is keying a value into the OTYPE field. Whatever that value is determines what subprocedure is called. You can combine a number of Ifs, Whens, and so on within the select.

```

C           SELECT
C* Maintenance type (Overhead or Underground)
C     OTYPE      WHENEQ    'M'
C           EXSR      ORDMNT
C* New Sales/Warranty type (Water Heater or Surge Protector)
C     OTYPE      WHENEQ    'N'
C     OTYPE      OREQ      'W'
C           EXSR      ORDWTR
C* Special type (Work Order or Pedestal)
C     OTYPE      WHENEQ    'S'
C           EXSR      ORDSPL
C           ENDSL

Select;
// Maintenance type (Overhead or Underground)
When OTYPE = 'M';
ORDMNT();
// New Sales/Warranty type (Water Heater or Surge Protector)
When OTYPE = 'N'
or OTYPE = 'W';
ORDWTR();
// Special type (Work Order or Pedestal)
When OTYPE = 'S';
ORDSPL();
EndSl;

```

Figure 10-11 Comparison of select statements

10.11 Substring and concatenate

Substrings and concatenates go hand-in-hand. A good developer always knows how to use them to their advantage and doing that in free-format is no different, and they are even more powerful in free format.

Figure 10-12 on page 376 is just one example of how you can use %Subst to your advantage. It follows the same rules as before. Lead with your result field (CHRORD) and then %Subst, followed by your factor and your length. In this program, and length is already defined, but just like old RPG, you can use numbers all the same. You can also use %Subst to change data types, such as pictured to the right. You can also use %Subst to do your old "MOVEL" command if you want.

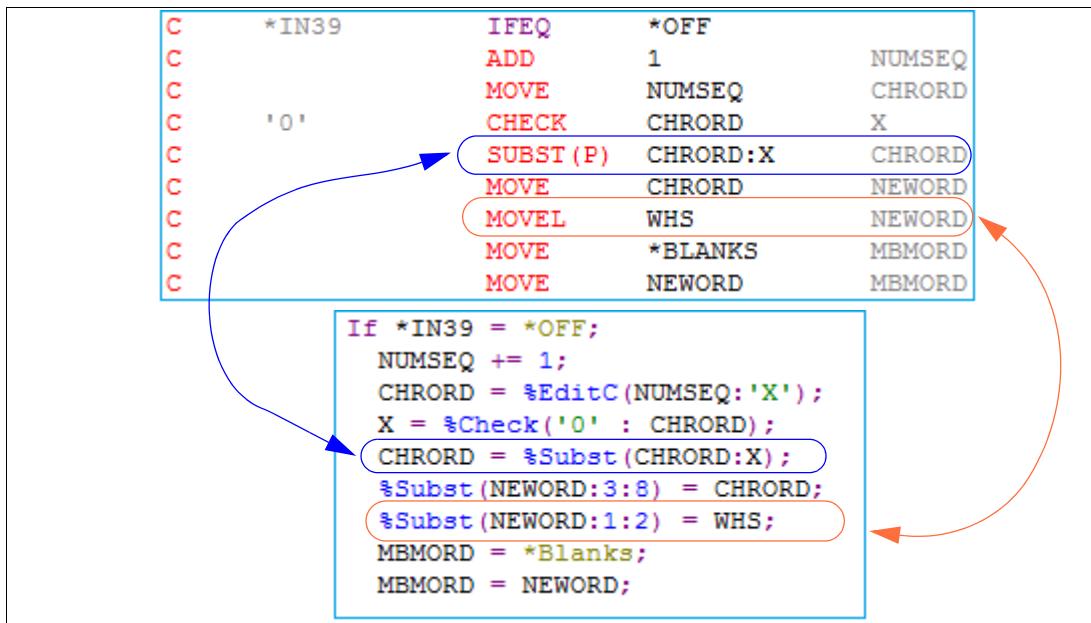


Figure 10-12 Comparison of substring

Concatenates in RPG free-format are even easier and make sense. Figure 10-13 show using concatenate in free-format. You do not need to add a "%" at the beginning of it. You just use the "+" sign. Your compiler knows if it is a character field or not, so it does not try to add it like it does on numeric.

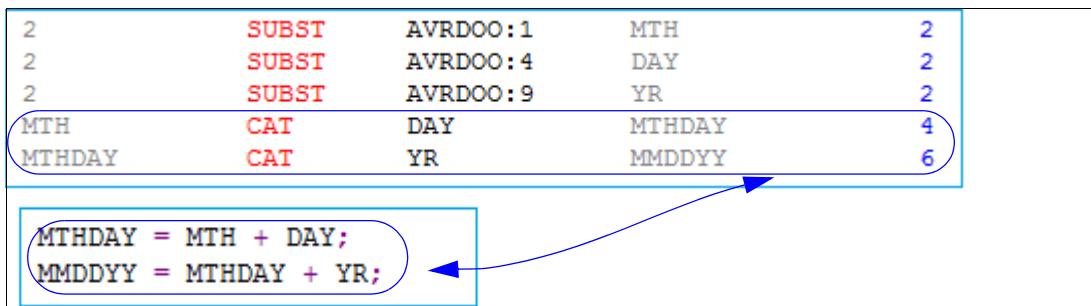


Figure 10-13 Comparison of concatenate

10.12 Go-To commands

Programming has become a lot more modular. Programmers know that using a Go-To command is just a way of being lazy, but it is so quick and easy that it is sometimes still used. In free-format, this command is not allowed, so you have to find some way of accomplishing the same outcome, just in a different way. While there might be a few ways to accomplish this, this is the option discussed in this section provides a way to do this.

You can still use the TAG command. You use the TAG command as a field to tell you when you need to skip over some code. In Figure 10-14 on page 377, the ATag command is used. In the fixed-format way, the tag returned to is named TSCR. Pretty descriptive, right? In all actuality, all we are doing is turning the Go-To command into a big DoU (Or DoW) loop. Once a condition is met that you want to jump over some code (or Go-To some code), you change the value of the tag you made, and it works the exact same way as a traditional Go-To

command. It may be a tad bit more work to do, but it's worth it to make your code more readable and modular.

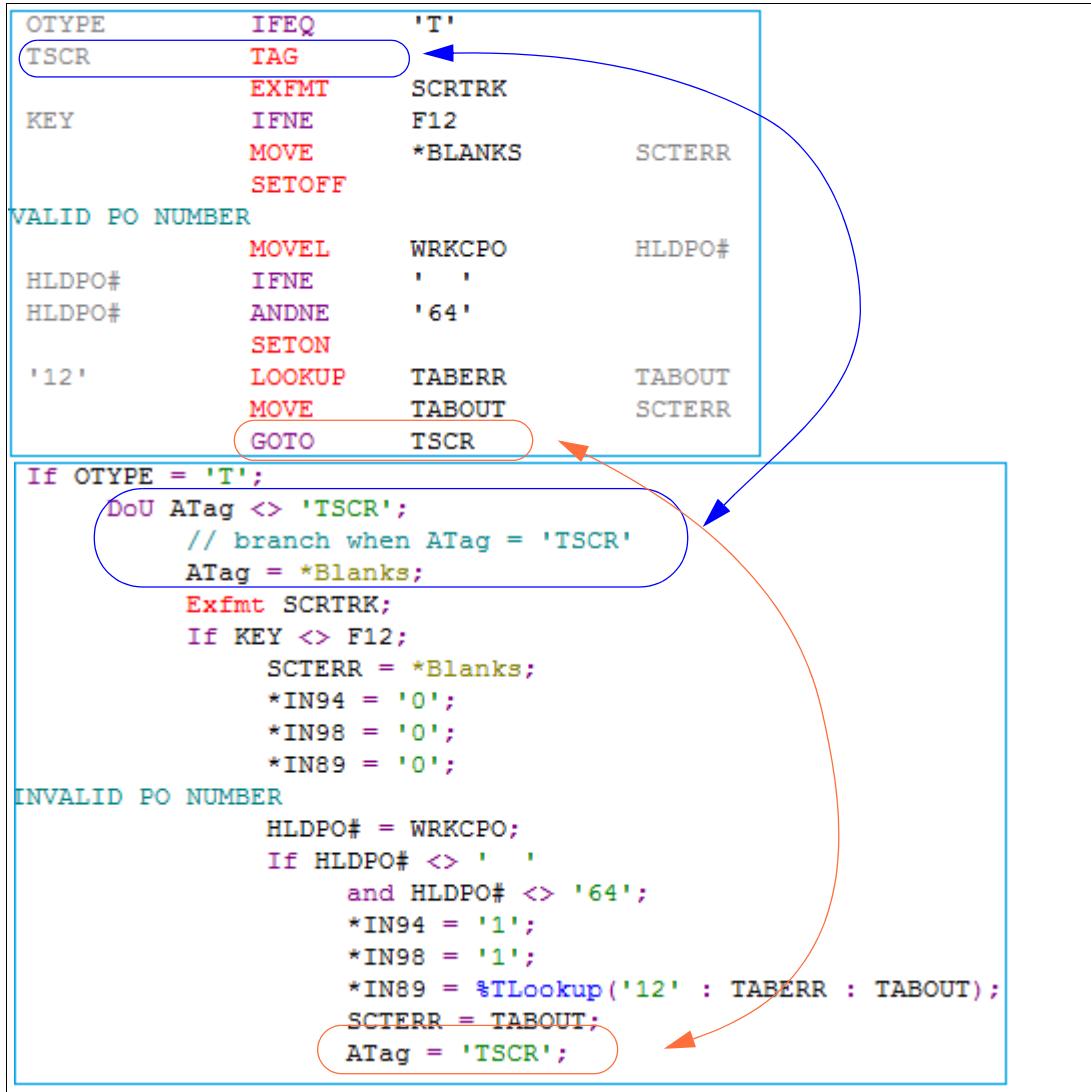


Figure 10-14 Comparison of Go-To command

10.13 Embedded SQL in free-format 101

While you can still use your typical read statements to get some data from a file, it might not be best to do that at all times. Depending on your time frame and how complex your program is, you as a developer might want to take advantage of embedded SQL instead. As with everything, there are many ways to accomplish a goal. This section discusses one of the quickest and easiest ways to get the data that you need leveraging some embedded SQL.

When reading in from a cursor, you need a structure to put your data in. These have to be unique, especially if you are reading in from more than one cursor, such as checking lower limits. There is no set naming convention to use, however you can use `SQL_` in the beginning to make sure to keep it readable.

Next you execute your SQL statement (Exec SQL) and declare your cursor. You can be as descriptive as you want throughout, but for this example Cursor1 and Cursor2 are used. If you want to be able to fetch all throughout your cursor (not just forward) you have to include the “scroll” keyword. This lets you get data from a variety of ways. You have to know a little bit about SQL in order to write your statement, but it is easy to learn and fits in with free-format RPG.

Table 10-1 list some SQL cursor actions.

Table 10-1 SQL cursor actions

Fetch Next	Positions the cursor on the next row of the result table relative to the current cursor position. NEXT is the default if no other cursor orientation is specified.
Fetch Prior	Positions the cursor on the previous row of the result table relative to the current cursor position.
Fetch First	Positions the cursor on the first row of the result table.
Fetch Last	Positions the cursor on the last row of the result table.
Fetch Before	Positions the cursor before the first row of the result table.
Fetch After	Positions the cursor after the last row of the result table.
Fetch Current	Does not reposition the cursor, but maintains the current cursor position. If the cursor has been declared as DYNAMIC SCROLL and the current row has been updated so its place within the sort order of the result table is changed, an error is returned.
Fetch Relative	Evaluates host-variable or integer to an integral value k. RELATIVE positions the cursor to the row in the result table that is either k rows after the current row if k>0, or k rows before the current row if k<0. If a host-variable is specified, it must be a numeric variable with zero scale and it must not include an indicator variable.
Fetch From	This keyword is provided for clarity only. If a scroll position option is specified, then this keyword is required. If no scrolling option is specified, then the FROM keyword is optional.

After you run your SQL statement, you open your cursor (Open Cursor1). Then you continue on fetching your data as needed. You might find it easier if you check your SQL commands against the SQL codes. If they pass those conditions, then go onto calculations and fetch the next record that you need.

You do not want to have to read all the records again if you want to do something special with your data. You can still check lower limits in SQL, it just requires an extra step and no keyword. Make another cursor (for example, Cursor 2) and then pass in your last record as a parameter (pictured to the right in Figure 10-15 on page 379). This keeps your runtime down and allows you to not start over each time you do some different calculations.

The rest is pretty simple and just follows basic RPG guidelines. Once you get the hang of embedded SQL, you will see how easy it is to get the data you need with minimal work, especially for reports.

```

Dcl-Ds SQL1;
// MKITRNHL9
// APVEND
// MKIMAST
SQL_MITVND Like (MITVND);
SQL_APVNAM Like (APVNAM);
SQL_MITITM Like (MITITM);
SQL_MITWHS Like (MITWHS);
SQL_MIMDES Like (MIMDES);
SQL_MITTDT Like (MITTDT);
SQL_MITQTY Like (MITQTY);
SQL_MITTRA Like (MITTRA);
SQL_MITTRP Like (MITTRP)
End-Ds;

BegSr StartSQL1;
// Get Records
Exec sql
  declare Cursor1 dynamic scroll cursor for
    SELECT ALL
      T01.MITVND, T02.APVNAM, T01.MITITM, T01.MITWHS,
      T03.MIMDES, T01.MITTDT,
      T01.MITQTY, T01.MITTRA, T01.MITTRP
    FROM MKTFILES/MKITRNH T01
    CROSS JOIN APCIFILES/APVEND T02
    CROSS JOIN MKTFILES/MKIMAST T03
    CROSS JOIN APCIFILES/APVMISC T04
    WHERE MITVND = APVNV#
    AND MITITM = MIMITM
    AND MITWHS = MIMWHS
    AND MITVND = APMVN#
      AND ( APMIMS <> ' '
        AND MITCD = 'R')
    ORDER BY T01.MITVND ASC, T01.MITITM ASC,
      T01.MITWHS ASC, T01.MITTDT ASC;

  Exec Sql
    Open Cursor1;

  Endsr;

// Fetch Records
Exec SQL
  Fetch Next From Cursor1 Into :SQL1;
  if sqlstt <> '02000';
  Exsr GetCost;
  Exsr StoreValues;
  endif;
  Exec SQL
    Fetch Next From Cursor1 Into :SQL1;
  EndDo;

  exec sql
    close cursor1;
  BegSr CheckLowerLimits;

  // Stores Current Values to Compare
  Exsr StoreValues;
  Exsr StartSQL2;
  // Check Records
  Exec SQL
    Fetch First From Cursor2 Into :SQL2;
    DoW SQLSTT <> '02000';
    Exsr PrintIt;

```

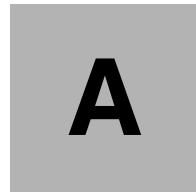
Figure 10-15 Calling SQL from free-format RPG basics

10.14 Built-in functions you need to know

There are a lot of built-in functions (BiFs) that IBM has added to the RPG language to help you get your jobs done. Table 10-2 on page 380 lists some of the more commonly used built-in functions. You can leverage the use of these functions to help build your applications with ease.

Table 10-2 Commonly used built-in functions

Function	Parameters & Code	Description
%CHAR	Graphic, date, time (or timestamp), numeric expression	Get value in character type.
%CHECK	Compare value : data-to-search (:StartPOS)	First position in data that contains a character not in the list of characters in the compare value.
%DATE(DateFormatCode)	(Value (Date Code if Needed))	Converting value to specified data format. System date is defaulted if not value is entered.
%DEC	Numeric expression (:digits :decPOS)	Value in packed numeric format.
%EDITC	Non-float numeric expression	String representing edited value.
%EOF	File Name	Represents End-Of-File without use of traditional indicators.
%Found	File Name	*ON if record is found in file (chain, delete, setgt, setll, and so on).
%LEN	Any expression	Returns the length of a variable or literal value.
%TLOOKUPxx	Search-data : searched table : alternate table	*ON if successful. Looks up data in table.
%MINUTES	Minutes	Duration value used to add a number of minutes to a time value.
%MONTHS	Months	Duration value used to add a number of months to a date value.
%OPEN	File name	Used to check if file is open (*ON or *OFF).
%REPLACE	Replacement string : source string (startPOS(SourceLengthTo Replace))	String produced by inserting replacement string into source string.
%SCAN	Search argument: string to be searched (startPOS)	First position of searched argument in string or zero (If not found).
%SUBDT	Date : duration code	Extracted component of the date value.
%SUBST	String:start(length)	Substring value.
%TIME	(value(:FormatCode))	Time data-type value after converting the "value" to specified time format.
%TIMESTAMP	Value:*iso	Timestamp data-type value.
%TRIM	String	String with left/right blanks removed.
%XLATE	From-table:to-table:string-to-convert:startingPOS	Converted string is returned.



Additional material

This book refers to additional material that can be downloaded from the Internet as described in the following sections.

Locating the Web material

The Web material associated with this book is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser at:

<ftp://www.redbooks.ibm.com/redbooks/SG245402>

Alternatively, you can go to the IBM Redbooks website at:

ibm.com/redbooks

Select the **Additional materials** and open the directory that corresponds with the IBM Redbooks form number, SG245402.

Using the Web material

The additional Web material that accompanies this book includes the following files:

<i>File name</i>	<i>Description</i>
REDBOOK.SAVF	IBM i save file containing the RPG code examples used in this book

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

IBM Redbooks

The following IBM Redbooks publications provide additional information about the topic in this document. Note that some publications referenced in this list might be available in softcopy only.

- ▶ *Modernizing IBM i Applications from the Database up to the User Interface and Everything in Between*, SG24-8185

You can search for, view, download or order these documents and other Redbooks, Redpapers, Web Docs, draft and additional materials, at the following website:

ibm.com/redbooks

Other publications

These publications are also relevant as further information sources:

- ▶ *ILE RPG Style Guide*
http://www.qrpglesrc.com/downloads/ILE_RPG_Style_Guide.pdf
- ▶ *Programming IBM Rational Development Studio for i, ILE RPG Reference*
https://www.ibm.com/support/knowledgecenter/api/content/n1/en-us/ssw_ibm_i_72/rzasd/sc092508.pdf
- ▶ *Programming in ILE RPG*, by Brian Meyers and Jim Buck
<http://www.mc-store.com>
- ▶ *Free-Format RPG IV*, by Jim Martin
<http://www.mc-store.com>
- ▶ *Programming IBM Rational Development Studio for i ILE RPG Programmer's Guide*:
https://www.ibm.com/support/knowledgecenter/ssw_ibm_i_72/rzasc/sc092507.pdf
- ▶ *Programming ILE Concepts*, SC41-5606
https://www.ibm.com/support/knowledgecenter/ssw_ibm_i_72/ilec/sc415606.pdf
- ▶ *DB2 for i SQL Reference*
https://www.ibm.com/support/knowledgecenter/en/ssw_ibm_i_72/db2/rbafzintro.htm

Online resources

These websites are also relevant as further information sources:

- ▶ IBM i Knowledge Center

- http://www-01.ibm.com/support/knowledgecenter/ssw_ibm_i/welcome
- ▶ IBM Training and Skills
<http://www-304.ibm.com/services/learning/ites.wss/zz/en?pageType=page&c=a0011023>
- ▶ IBM developerWorks RPG Cafe
<https://www.ibm.com/developerworks/community/groups/service/html/communityview?communityUuid=b542d3ac-0785-4b6f-8e53-f72051460822>
- ▶ Search400
<http://www.search400.com>
- ▶ Midrange.com website provides a large number of resources for users and developers of IBM i
<http://www.midrange.com>
- ▶ The IBM i user group, COMMON
<http://www.common.org>
- ▶ iProDeveloper:
<http://iprodeveloper.com/>
- ▶ ARCAD Transformer
<http://arcadsoftware.com/products/arcad-transformer-ibm-i-refactoring-tools/>
- ▶ Linoma Software RPG Toolbox
<http://www.linomasoftware.com/products/rpg-toolbox>
- ▶ System i Developer's RSE Quick Start Guide
<http://systemideveloper.com/downloadRSEQuickStartGuide.html>
- ▶ RDi Trial download
https://ibm.biz/rdi_download
- ▶ Online learning resources from IBM
https://ibm.biz/rdi_wiki_self_learning
- ▶ System i Developer's Favorite Shortcut Keys card
<http://systemideveloper.com/downloadRSEShortcuts.html>
- ▶ A Closer Look at iSphere
<http://www.ibmsystemsmag.com/ibmi/developer/rpg/isphere-details/>
- ▶ iSphere Plug-in Expands RSE/RDi Toolset
<http://www.itjungle.com/fhg/fhg061615-story01.html>
- ▶ More iSphere Goodies
<http://www.itjungle.com/fhg/fhg092915-story03.html>
- ▶ Looking For Stuff With iSphere
<http://www.itjungle.com/fhg/fhg070715-story01.html>
- ▶ iSphere Plug-in on SourceForge
<http://isphere.sourceforge.net/>

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

To determine the spine width of a book, you divide the paper PPI into the number of pages in the book. An example is a 250 page book using Plainfield opaque 50# smooth which has a PPI of 526. Divided 250 by 526 which equals a spine width of .4752". In this case, you would use the .5" spine. Now select the Spine width for the book and hide the others: **Special->Conditional Text->Show/Hide->SpineSize[-->Hide]->Set**. Move the changed Conditional text settings to all files in your book by opening the book file with the spine.fm still open and **File>Import>Formats** the Conditional Text Settings (ONLY) to the book files.

Draft Document for Review October 20, 2016 11:51 am

5402spine.fm 387



Who Knew You Could Do That with RPG IV?

SG24-5402-01
ISBN DocISBN



(1.5" spine)
1.5" <-> 1.998"
789 <-> 1051 pages



Who Knew You Could Do That with RPG IV?

SG24-5402-01
ISBN DocISBN



(1.0" spine)
0.875" <-> 1.498"
460 <-> 788 pages



Who Knew You Could Do That with RPG IV? Modern RPG for the

SG24-5402-01
ISBN DocISBN



(0.5" spine)
0.475" <-> 0.873"
250 <-> 459 pages



Who Knew You Could Do That with RPG IV? Modern RPG for the Modern

(0.2"spine)
0.17" <-> 0.473"
90 <-> 249 pages

(0.1"spine)
0.1" <-> 0.169"
53 <-> 89 pages

To determine the spine width of a book, you divide the paper PPI into the number of pages in the book. An example is a 250 page book using Plainfield opaque 50# smooth which has a PPI of 526. Divided 250 by 526 which equals a spine width of .4752". In this case, you would use the .5" spine. Now select the Spine width for the book and hide the others: **Special>Conditional Text>Show/Hide>SpineSize[-->Hide]>Set**. Move the changed Conditional text settings to all files in your book by opening the book file with the spine.fm still open and **File>Import>Formats** the Conditional Text Settings (ONLY) to the book files.

Draft Document for Review October 20, 2016 11:51 am

5402spine.fm 388



Who Knew You Could Do That with RPG IV?

SG24-5402-01
ISBN DocISBN



(2.5" spine)
2.5" <-> nnn.n"
1315<-> nnnn pages

Who Knew You Could Do That with RPG IV? Modern RPG for the Modern

SG24-5402-01
ISBN DocISBN



(2.0" spine)
2.0" <-> 2.498"
1052 <-> 1314 pages



Draft Document for Review October 20, 2016 11:52 am



SG24-5402-01

ISBN DocISBN

Printed in U.S.A.

Get connected

