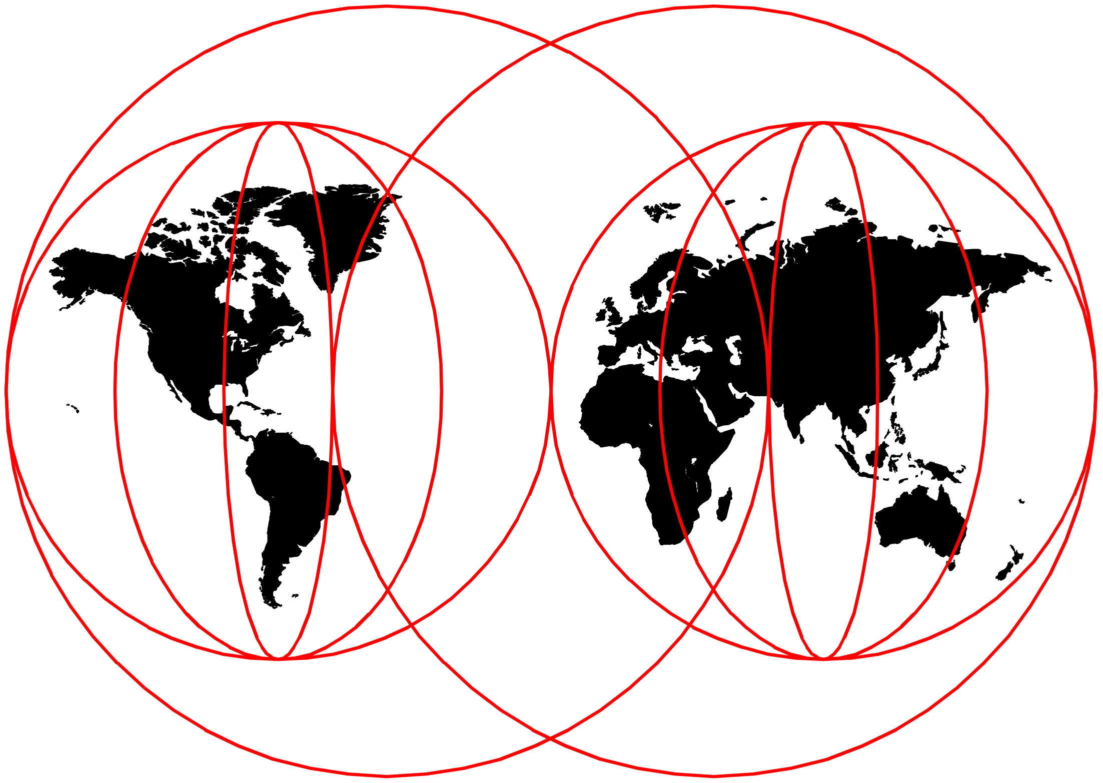
Who Knew You Could Do That with RPG IV? A Sorcerer's Guide to System Access and More

Brian R. Smith, Martin Barbeau, Susan Gantner,
Jon Paris, Zdravko Vincetic, Vladimir Zupka



International Technical Support Organization

www.redbooks.ibm.com

SG24-5402-00

International Technical Support Organization

SG24-5402-00

Who Knew You Could Do That with RPG IV? A Sorcerer's Guide to System Access and More

February 2000

Take Note!

Before using this information and the product it supports, be sure to read the general information in Appendix D, “Special notices” on page 419.

First Edition (February 2000)

This document was updated on April 25, 2000.

This edition applies to Version 4 Release 4 of the IBM Integrated Language Environment (ILE) RPG for AS/400 (5769-RG1) for use with Version 4 Release 4 of OS/400 (5768-SS1).

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. JLU Building 107-2
3605 Highway 52N
Rochester, Minnesota 55901-7829

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Figures

Tables

Preface

This redbook is focused on RPG IV as a modern, thriving, and rich application development language for the 21st century. It is written for those AS/400 system programmers that are in the cusp between RPG/400 and RPG IV and are looking for hints and tips to make the move forward worth their while. This book promises to drop little golden nuggets of information in the form of code samples and style guidelines. Picking up each golden nugget will lead you step-by-step down the path that will eventually allow you to take full advantage of RPG IV and the Integrated Language Environment (ILE). Even the most experienced RPG IV programmer will find something useful in this redbook.

Update notice

The SSERVER3 and SCLIENT3 example program as an example of multiple I/O between one server and multiple clients was not correct and has been updated. The updated sections are marked by change bars in 5.5.5, "Communication with multiple sockets (multiple I/O)" on page 210 and 5.5.6, "Example of multiple I/O" on page 215.

In addition, we updated the RPGISCOOL library SAVF found under Additional materials on the IBM Redbooks Web site. See Appendix A, "Example RPG IV programs on the Web" on page 411, for instructions to download these examples. The source files in library RPGISCOOL file SCKSRC that have been updated are:

- SCKSELF
- SCKCPY
- SCLIENT3
- SSERVER3
- SCLIENT3B
- SSERVER3B

We know you have been busy running your business instead of following RPG IV enhancements. So, we have included a timeline outlining the history of the RPG IV language and all the technical updates to the language since V3R1. You can use this timeline to assist in determining which of the many enhancements you can take advantage of to solve the real business problems today.

In V3R2 and V3R6, RPG IV added support for user-defined subprocedures marking the defining point at which RPG IV can truly be considered a modern programming language. This redbook shows you both the style and function of how to make the subprocedures work for you.

One of the keys to the power of RPG IV is in its ability to prototype any system function and make things happen! This redbook shows you how to use RPG IV to:

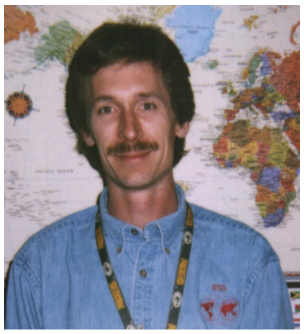
- Use the TCP/IP sockets APIs
- Read and write directly to the Integrated File System (IFS)
- Use dynamically server HTML Web pages with the CGI interface
- Exploit program-to-program communications with data queues (DTAQ)
- Directly access user spaces (USRSPC)
- Communicate with the system and users via message handling
- Write exit programs (anonymous FTP)

RPG IV and DB2/400 have always enjoyed a tight integration between language and database. Now, learn how to exploit the new CLI database APIs and access your business data with embedded SQL, stored procedures, trigger programs, and commitment control (CC).

Modern tools can significantly improve your productivity. This redbook also briefly introduces such tools as CODE/400 for rapid server-side development and VisualAge for RPG for rapid client-side RPG in a client/server environment.

The team that wrote this redbook

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



Brian R. Smith is a Senior AS/400 Specialist in the International Technical Support Organization in IBM Rochester. The first half of his career was spent in design, coding and testing on the System/38 and AS/400 in the area of communications. He then "jumped the wall" into technical marketing support in 1990 to pursue the life of teaching and writing.

You can reach Brian on the Internet at `brsmith@us.ibm.com`

Martin Barbeau is a Software Support Specialist in Canada. He has 10 years of experience working with AS/400 systems and has worked for IBM for three years. His main areas of expertise include system management, application development (mainly using RPG and DB2/400), availability/recovery and OS/400 problem determination. Martin is the co-author of the IBM redbook Complementing AS/400 Storage Management Using Hierarchical Storage Management, SG24-4450.

Susan Gantner has spent over 20 years in the field of application development. She has recently joined Hal North America as a Senior Consultant for the Education Group following a 14-year IBM career. Prior to IBM, Susan developed applications for corporations in Atlanta, GA, working with a variety of hardware and software platforms. In 1985, she joined IBM as a Systems Engineering Specialist for the System/38. She worked in the Rochester Lab for several years supporting and teaching AS/400 customers in the areas of application development and database. She later moved to the IBM Toronto laboratory to continue her teaching and customer support roles for the languages and development tools. She is a regular speaker at COMMON and technical conferences for AS/400 customers around the world. She can be reached at `susan.gantner@halinfo.it`

Jon Paris is a Senior Consultant with the education group of Hal North America, based in Toronto, Canada, where he specializes in AS/400 Application development education. Jon has accumulated over thirty years experience in the computer systems field. He has also published a number of articles on AS/400 application development.

Prior to joining Hal, Jon worked for the IBM Toronto Laboratory where, among other things, he was one of the original "fathers" of the RPG IV language. Subsequently, he worked with the Lab's Application Development Market Support team. In this role, he was responsible for producing educational, support and services materials for use throughout the world. Jon's area of expertise was in the AS/400 programming languages and development tools, including CODE/400, VisualAge for RPG, RPG IV, COBOL, and VisualAge for Java. Jon presented on these and other related topics at User Group meetings, IBM conferences, and other events around the world. He was also active in preparing and delivering education to IBM Business Partners on the latest trends in AS/400 Application Development. You can reach him by e-mail at `jon.paris@halinfo.it`

Zdravko Vincetic is an AS/400 Systems Specialist at the AS/400 Systems Support Center in IBM Slovenia. He has 27 years of experience in the IBM S/3, S/38, and AS/400 field, all with IBM, and spent most of this time teaching IBM classes. His areas of expertise include application development, database design and systems management. He holds a Bachelor of Science degree in Mechanical Engineering from University in Sarajevo and a Masters degree in Informatik from University in Zagreb.

Vladimir Zupka is a freelance consultant and programmer providing services for IBM AS/400 systems in Czech Republic and other countries. He passed state examinations on the Czech Technical University (CVUT) in Prague as a land-surveyor engineer. He has 34 years of experience in programming on various computers from relay, bulb, or transistor based machines through Univac 9300 and IBM 370 to AS/400. His areas of expertise include system programming in machine, assembler, and high-level languages, compiler development, and business application programming. He published a book on RPG II programming in 1985 in Prague. He has been giving lectures about the AS/400 system, especially programming in RPG language. He became an IBM Business Partner in 1997. You can reach him by e-mail at `vzupka@comp.cz`

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

Paul Tuohy
ComCon, AS/400 consulting company - CEO

Simon Coulter
FlyByNight Software Australia - AS/400 Software Engineer

Ivy Woo
Hal North America Limited

Ed A Van Weeren
IBM Netherlands - BAAN/IBM International Competency Center

Yvonne Griffin
IBM Rochester Lab

John Kasperski
IBM Rochester Lab Transport Services - Sockets

Mark Megerian
IBM Rochester Lab - CLI Database and Threads

Mel Rothman
IBM Rochester AS/400 Custom Technology Center - AS/400 RPG, AS/400 CGI Programming

Hans Boldt
IBM Toronto Canada Lab - ILE RPG Development

Barbara Morris
IBM Toronto Canada Lab - RPG Compiler Development

Karan Ratz
IBM Toronto Canada Lab - RPG CPS Development

Claus Weiss
IBM Toronto Canada Lab - VisualAge RPG Development

Brian Parkins
IBM UK Education & Training - AS/400 Instructor

Bob Cozzi
Independent Consultant

Sandro K. Pires
Itec S/A Brazil - System Engineer, AS/400 Technical Support Group

Bryan Meyers
400 School

Comments welcome

Your comments are important to us!

We want our redbooks to be as helpful as possible. Please send us your comments about this or other redbooks in one of the following ways:

- Fax the evaluation form found in “IBM Redbooks evaluation” on page 433 to the fax number shown on the form.
- Use the online evaluation form found at `http://www.redbooks.ibm.com/`
- Send your comments in an Internet note to `redbook@us.ibm.com`

Chapter 1. An introduction to RPG IV

The RPG IV language was introduced in Version 3 Release 1 (V3R1) of OS/400 and is still a relatively new language. For a complete history of the enhancements made to the RPG IV language, see 1.3, “Evolution of the RPG IV language” on page 9. This section also includes a quick look at the future of RPG and of the RPG programmer on the AS/400 system.

A common source of confusion to the RPG/400 programmer is the relationship between RPG IV and the Integrated Language Environment (ILE). ILE was introduced to the AS/400 in Version 2 Release 3 (V2R3). Moving to RPG IV does not necessarily require the use of, or the understanding of, the Integrated Language Environment (ILE). See 1.4, “The relationship between the RPG IV language and ILE” on page 14, for an explanation.

In today’s business environment, certification is becoming increasingly important and RPG certification is no exception. Find out more in 1.5, “IBM Certified Specialist AS/400 RPG programmer” on page 16.

1.1 Why this book on RPG IV now?

The last redbook that specifically focused on RPG was published in April 1995. This book is called *Moving to Integrated Language Environment for RPG IV*, GG24-4358. It is still in print and averaged 1500 download *hits* per month in mid-year 1999! We have all learned so much about how to write RPG IV applications in the past four years that it was time to take the coding pen in hand and show off what can be done with this updated language.

Other redbooks have followed. In September 1996, IBM delivered an application development tools redbook entitled: *AS/400 Applications: A Fast and Easy Way to Install, Set Up and Work with VRPG and CODE/400 (ADTS CS)*, SG24-4841. This redbook did a great job on just what the title says it does. In fact, it is still a great guide to VisualAge RPG and CODE/400.

Then, in January 1997, we published another redbook on application development that focused on the Y2K (Year 2000) problem: *AS/400 Applications: Moving to the 21st Century*, SG24-4790. Most of the application programming examples found in this book were written in RPG.

Is RPG IV a dead language? The answer to this question cannot be as simple as yes or no. The truth to the matter lies in the fact that all languages rise in popularity, plateau, and then over time, decline. This redbook provides examples that allow you and your team to extend that plateau by making most of what you have now. Nobody said that RPG’s plateau should not last for a thousand years!

1.2 An interview with leading industry experts

"Rumors of my death have been greatly exaggerated"

While this phrase originated with Samuel Clemens (also known as Mark Twain), it could equally well be said of the RPG programming language. We have been hearing the cry "RPG is dead" for years. Yet here we are at the beginning of the 21st century and RPG remains with us, as strong as ever. Why is this?

We decided to talk to a number of AS/400 experts to get their opinions. First, we introduce you to each of our experts:

- **John Carr** is President of EdgeTech, a Virginia corporation. He has 17 years of experience working with the IBM Midrange Environments. His expertise lies in helping companies develop and implement programming standards and methodologies.

  John has aided clients in improving their proficiency and efficiency with the Midrange Environments. In this regard, John has taught numerous classes for EdgeTech focused on application system development including RPG/400, SQL/400, ILE, ILE/RPG, VRPG, and Relational Database to name a few.

  John has served on various IBM Advisory Councils concerning ILE/RPG, VRPG, and the Future of the AS/400 system. He is currently working with IBM on a Certification program for RPG. He has written articles for COMMON's NEWS COM magazine on the transition from RPG to OOPs and has been a popular speaker at COMMON for the past 7+ years.
- **Simon Coulter** is a principal of FlyByNight Software, an AS/400 consulting company based in Melbourne, Australia. He specializes in technical programming and AS/400 programming education. He is currently involved in an AS/400 project using sockets and Telnet programming. Simon was an AS/400 Software Engineer with IBM Australia for eight years and has 13 years of experience in the computer industry. He is a regular presenter at COMMON Australasia and helped produce the redbook *Building AS/400 Applications with Java*, SG24-2163.
- **Bob Cozzi** has been involved in the midrange IT community since 1977. He has been a member of COMMON, the IT industry's largest user group, since 1980. He has worked with several end-user firms and has run his own small business. In addition, Bob published the midrange industry's first technical magazine, *Q38 Technical Journal* and later *Midrange Computing* magazine. In 1988, he wrote and published the first book on RPGIII programming *The Modern RPG Language* and has since written five additional titles including the popular *The Modern RPG IV Language* book, which along with the original title continues to be the best selling midrange book ever published.

  In 1995, Bob left his publishing and computer research firm to pursue other interests. Since then he has acquired an interest in Pioneer Rocketplane which is building the next generation space shuttle. He continues to maintain a strong interest in RPG and the AS/400 system. He owns and maintains the RPG Developer Network, a Web site exclusively for RPG programmers (`http://www.rpgiv.com`), and lectures on RPG IV at customer locations throughout North America.
- **Bryan Meyers** has been a longtime Senior Technical Editor for *NEWS/400* magazine. Most AS/400 technicians know Bryan from his regular articles appearing in *NEWS/400*, and his books and speaking tours. He is the author of the books *RPG/IV Jump Start*, *Starter Kit for the AS/400*, *Control Language Programming for the AS/400*, *VisualAge for RPG by Example*, and several others.

  Bryan presents *NEWS/400* Technical Seminars across North America, on many topics including RPG IV, VARPG, and the Integrated Language Environment (ILE). Bryan is the Vice President of Education for the Powertech Group, heading its 400 School division.

- **Russ Popeil** is the Northeast Area Consulting Practice leader for Avnet Computer, a national IBM business partner, Russ is based in New York City and has over 18 years experience on the IBM midrange platform. He is a past President of the Long Island Systems Users Group and a frequent speaker at COMMON, and other local user groups.

  Russ is the author of the book *RPG Error Handling Technique: Bulletproofing your applications*. He has also written many articles for *NEWS/400* and *Midrange Computing* magazines.

  Russ is currently working with IBM as a member of the IBM certification test writing teams, for RPG and AS/400 Technical Solutions. He is also on the IBM Toronto language labs RPG advisory council and a member of the IBMs J.D. Edwards competency center business partner internship team.

- **Paul Tuohy** has worked in application development on IBM midrange for 20 years. He worked as a programmer/analyst for five years before taking on the role as IT manager for Kodak Ireland. After three years, he returned to application development and became Technical Director of Precision Software, a successful Irish software company specializing in applications in Shipping and Human Resources. After eight years with Precision Software, he set up ComCon, an AS/400 Consultancy company. Paul now spends most of his time presenting AS/400 courses. Generally, he is located at the IBM Education Center in Dublin, Ireland, where he has been specializing in courses in programming and application development since 1988.

  Paul is the author of the article "Re-engineering AS/400 Legacy Applications", published by Midrange Computing.

And so our interview begins ...

How old are you?

John: I'm 47

Simon: 36

Bob: 40

Bryan: Old enough to make that a rude question ... 51

Russ: 43

Paul: 18 going on 43

Was RPG your first programming language?

John: Yes.

Simon: No, I first learned Basic and 6502 Assembler on a BBC Micro (self-taught), then RPG II (1 week training, no practice), then Assembler and INFORM (a 4GL of sorts) on an ICL System 25 (1 week of training, six months or so of practice, then RPG III on System/38, self-taught MI, PL/I, COBOL, FORTRAN, C, Smalltalk, and now Java (also some dabbling with AWK, Unix scripts, BAT files, CMD files, and Rexx along the way). The language is immaterial; you're either a programmer or you're not.

Bob: First learned, COBOL, first used, RPG.

Bryan: Yes, unless you count JCL.

Russ: Yes.

Paul: No. My first "real" language was DIBOL. This was on DEC equipment and was like a cross between Basic and COBOL.

When did you write your first RPG program, and what version of the language?

John: 1984 RPGII on a System/34,

Simon: RPG II on a Control Data Cyber something-or-other as part of my training. Although, first use in anger for a business application was RPG III.

Bob: 1977 in RPGII for the System/32.

Bryan: I started with an early version of RPG II. Tables were a fairly new construct at the time.

Russ: 1982 in RPGIII on the System/38.

Paul: Around 1979 on the System/34. So it would have been some form of RPG II.

What did your first RPG program do?

John: A report, what else?

Simon: Simple reporting in RPG II. The first RPG III application was an interface between JDE general ledger on S/38 and StarBase financial application (AP/AR) on ICL Sys25. This is where I discovered I was better suited to systems work rather than application work.

Bob: Print labels for packages distributed to McDonalds stores, and it is still in use today.

Bryan: During my college days, I worked as a disc jockey at a local radio station at one of the first broadcasting companies to use a computer to schedule programs and commercials. Most of my work was in that area. The first program I specifically remember sorted, selected, and listed Top 40 hit songs that were keypunched onto 80 column cards.

Russ: I worked at a hospital and we printed a report of device locations.

Paul: Heck if I can remember. I am sure the word "print" came somewhere.

Is RPG dead? If so, why? If not, why not?

John: Ah Hmm. It's as dead as COBOL. And it's as dead as it ever was.

Simon: No, it's not! For the same reasons as COBOL not dying. Too much investment in existing code compounded by lack of interest in better methods by "grunt" RPG programmers.

Bob: No, it is still the only business-oriented language that is in wide-use on the AS/400. Existing applications will need to be enhanced and maintained for another 20 or more years. In addition, it provides a solid, reliable back-end language to GUI-based tools, such as Java.

Bryan: I find it hard to believe that billions of lines are code are going to fade away easily anytime in the foreseeable future. RPG programs will coexist with other languages in the IBM midrange for some time to come, I think.

Russ: It is not dead and won't be for a very long time. There is too much invested in existing RPG code and most shops do not want to be multi-lingual.

Paul: A definite *no*! For RPG to die, one of two things has to happen. Either the AS/400 is discontinued or companies decide to completely re-write their applications in another language. The former is highly unlikely (although IBM has surprised me in the past), and it would be extremely difficult to make a business case for the latter. The impact of a language like Java will impact the interactive side of RPG. The days of the green screen may be numbered, but I think one would be hard pushed to justify re-writing all of the business rules, etc.

What percentage of your RPG work is done in RPG IV? If the answer is less than 100%, why?

John: 100% since V3R1. During the Y2K conversion, we converted *all* inhouse programs to RPG IV, along with using *all* Date/Time data types in our files.

Simon: 25% because I don't do application programming. Most of my time is spent writing systems code and the usual choice is C, not because I particularly like C, but it is more acceptable as a systems language. I use RPG IV for utilities or programs for sites that only have RPG. I keep my hand in by teaching RPG for IBM Learning Services.

Bob: 100%

Bryan: Less than 100%. Why? All my new RPG work is done in RPG IV.

Russ: I write 100% in RPG IV.

Paul: 120%. The additional 20% is because I also use ILE. I can't see any reason for working on an earlier version of RPG.

What was the biggest problem that RPG IV solved that you could not have accomplished with older versions of the language?

John: Where do I start? Interfacing with UNIX APIs and C APIs. Writing to the IFS. Doing Date Math. Otherwise, it makes things *tons* easier, faster, clearer, etc. Triggers are simpler. There is no 256 field length limit, for example. Softcoding functions using BIFs (Built-in Functions) like %LEN, %ELEM, for example, cuts down on maintenance overhead. Procedures, prototyping, writing our own "Opcode" like functions.

Simon: Encapsulation! Locally scoped data and parameter passing to procedures are worth their weight in gold. Of course, sufficiently disciplined programmers could accomplish the same result with RPG III. But RPG IV makes it easier for someone who knows what they are doing to create black boxes for the rest of the group to use.

Bob: Gave us the ability to use contemporary programming techniques.

Bryan: Being able to use functions and write procedures that I can easily incorporate into my programs has made many programs more modular, flexible, and maintainable. I think that's the single biggest improvement in the language.

Russ: Using the evaluations operation allowed string handling to be a breeze.

Paul: I don't know if RPG IV solved any "problems". What it did is make viable programs that would not have been previously considered due to the length of time that it would take to develop them. This is mainly due to ILE and subprocedures.

What do you think are the best features of RPG IV? What is the worst?

John: The best are the SIMPLE I/O functions. One F-Spec for an externally defined file would equal how many in other languages? The worst are mostly things outside the language itself, for example, lack of a GUI isn't RPG's problem exactly.

Simon: Free-form expressions, built-in functions, procedures, and the PREFIX keyword are the best features. The worst is probably continued support for archaic coding methods, for example, conditioning indicators, resulting indicators, COMP, CMPxx, and GOTO. LEAVE, ITER, and LEAVESR are in there too, but are more tolerable than GOTO.

Bob: The best is Procedures. Unnecessary "tick marks" such as the COLON, I would have used either a blank (to make it the same as DDS) or a comma. The "E" on externally defined data structures is another example.

Bryan: D-specs, expressions, built-in functions, and of course procedures are the best features. It's difficult to find a worst feature, but still forcing tables to start with TAB seems unnecessary. Putting compiler options in the source is a good idea, but it's implemented in a half-hearted fashion. Missing features include an increment/decrement operator.

Russ: The best feature is Procedures. The worst is?

Paul: The two I have found most useful are mixed case (believe it or not) and subprocedures. Any of the bad features can just be ignored.

Do you have plans or have you already undertaken a Java, C, or C++ development project?

John: Java, no, not yet. but it's coming. I guess after the first of the year, mostly in conjunction with our Lotus Domino applications at first I think. On the other hand, there are no plans at all for C.

Simon: I've done a few projects in C. C++ is a waste of time. If you really want to do OO, use Smalltalk. Java is on the cards, but not much is happening yet in Australia with AS/400 Java. When I can take time out from earning a living, I'll implement some of the ideas I have for AS/400 Java.

Bob: Have already done this.

Russ: No plans yet.

Paul: Have played around with Java, and a Java project is planned.

Can you name the individual or group that has the most impact on your professional life?

John:	Early hard bosses. "Do it right the first time!!!"
Simon:	Not one individual, but more of a gestalt of various teachers and colleagues and of course, eight years of writing system code for IBM has had a major influence on the way I engineer software—people like Bill Davidson and Dick Bains at Rochester. Bob Cozzi and Greg Veal have also had some influence on my programming. I subscribed to the Q38 Journal, and most of what they said made sense. P. J. Plauger, Donald Knuth, Kent Beck, and others also.
Bob:	IBM, unless you mean groups like COMMON, then it is COMMON.
Russ:	COMMON and LISUG (Long Island AS/400 User Group).
Paul:	Too many to mention them all. As a group, the students that attend my courses continue to teach me. I have worked with some very good technical people in Precision Software, and the best teacher I have had is Jon Paris.

If you had the power to add one feature to RPG IV. What would it be?

John:	Well I have gotten most of the things I have lobbied for over the last few years.
Simon:	User-defined data types. These would improve robustness of interfaces by allowing prototyping to verify correct structure types are being passed to procedures and programs.
Bob:	Keyword support on the Output specifications.
Bryan:	I like to see an increment (++) and a decrement (--) operator for expressions.
Paul:	Can't choose just one, although the ability to define keylist on the D-Spec would probably edge the others out. All that are left are fairly minor. The major ones have already been, or are about to be, incorporated. Full free format would be my major requirement, but the new CF spec will take care of that.

If you were IBM, what would you do to encourage more people to switch to RPG IV?

John:	Reference the language in their publications and announcements. Every thing spoken about RPG from IBM gives the tone that they are ashamed of the language and just want to have everyone forget it. Develop a strong affinity with RPG IV and Java. Provide a native GUI for RPG.
Simon:	Announce end-of-support for RPG II, III, and 400. Brute force but probably the only way that'll work, unless we get some good marketing that makes RPG IV appear appealing. But IBM can't market, so that won't happen. If they could, we'd all be running OS/2 instead of suffering windoze (although I'm sticking with OS/2). It would also help if IBM encouraged the major ERP vendors to start using RPG IV, but they are more likely to jump into Java or use C simply to gain cross-platform benefits from any major application re-work.
Bob:	Make them want it, and then make them not be able to live without it. That is, stop enhancing it with an apparent attitude that you're

doing us a favor (and start pushing Bob Cozzi's RPG IV book more!). It needs to be pushed as solving solutions for contemporary problems. Like a back-end to a Java front-end. Instead, we're seeing "Java, Java, Java" without a goal other than to get people to use Java. We (RPG programmers) have seen and heard this before, and we just don't buy into it. So we end up feeling left-out once again. And yet, our little RPG language keeps running and running. Remember when everybody was pushing Visual Basic, then C++ for a month or two? If we can give managers clear examples of RPG IV being used to solve their problems, then they will give the okay to use it.

Bryan: I'd give them a clear direction on the future of the language, and explain how it will fit into their future application development. I'd quit trying to scare them into thinking that RPG is a dead end for their careers. I'd make them understand that they don't need to (and probably shouldn't) abandon one language to begin taking advantage of another.

I'd also take every opportunity to have RPG taught in schools, colleges, and private training facilities. Rather than forming an entirely new separate education structure, take advantage of the facilities that already exist, and that already have students. Give them support, give them hardware, give them software, give them class materials, and give them encouragement. The investment would be well worth it in the long run.

Russ: Advertise the ease that one can get up and running in RPG IV, along with the productivity enhancements that will benefit the users of RPG IV.

Paul: Make it crystal clear that RPG IV does not mean flipping burgers. Identify it as part of the road to Java. Incorporate the tools to do the work. Code/400 should be incorporated as a standard part of Application Development Tools.

1.3 Evolution of the RPG IV language

The RPG IV language still relatively young, but it has grown dramatically in the few years since its introduction.

Examples in this book are based on V4R4

The RPG programs and AS/400 database libraries used in this redbook are available for you to download from the Internet. These examples were developed using an AS/400 system with OS/400 and the RPG compiler (5769RG1) at V4R4. See Appendix A, “Example RPG IV programs on the Web” on page 411, for instructions on how to download the source.

Some of the programming examples use functions and features of the compiler and OS/400 system that will only compile and run at V4R4 of OS/400. Others may run on a V3R2 or V3R7 system and above with or without minor modifications. Use this section to help identify those enhancements to the RPG IV language that may prevent you from simply compiling the example programs without any modifications.

1.3.1 Version 3 Release 1 (V3R1)

The RPG IV language was introduced at Version 3 Release 1 (V3R1) of OS/400. At its first release, the new RPG language represented a significant improvement over the previous RPG languages. The following list contains some of the most significant enhancements in the first release:

- Expanded names (to 10 characters)
- Mixed case source
- Free-form expressions (arithmetic, string and logical expressions)
- Support for date and time data types, including duration calculations
- A new D specification to define all types of internal program data
- Built-in functions
- Many previous language limitations raised or removed (Some examples would be the maximum number of files and the maximum size of arrays and data structures.)

1.3.2 Version 3 Release 2 (V3R2) and Version 3 Release 6 (V3R6)

The next release of the RPG IV compiler was at V3R2 and V3R6. Both of these releases contained an equivalent RPG IV compiler function. As significant as the new language syntax and features were at V3R1, this second release introduced the potential for an even more dramatic change to the structure of modern RPG IV programs. The introduction of prototyping and the ability to write RPG IV subprocedures changes the way RPG IV programs may look in the future. The ability to prototype calls to functions in the operating system expands the capabilities of RPG programmers to include the entire C language function library and other system functions previously only available to C programmers.

The following list contains the most significant enhancements in this release. While the number of items seems small in comparison to other releases, the significance is far greater than any other RPG IV compiler release to date.

- User defined RPG IV subprocedures
- Prototyping for call interfaces, including support for return values from functions
- Free-form CALLP operation code (Call with Prototype)
- Cycle-free option for modules containing only subprocedures

1.3.3 Version 3 Release 7 (V3R7)

At V3R7, the RPG IV compiler grew yet again in function. Note that V3R7 (and later) enhancements can only be used on AS/400 RISC systems. While perhaps less significant in the grand scheme of things, this list of enhancements represents the fulfillment of many long-standing requirements and ease of use features:

- Null value support for database fields
- Null terminated string support
- Floating point data type
- Better control over precision in free-form expressions
- New built-in functions for string editing (%EDITC and %EDITW)
- %SCAN built-in function
- Allocate and deallocate storage by operation code
- Ease of use enhancements for date and time fields
- Conditional compilation directives embedded in the source
- Nesting support for the /COPYstatement
- Longer names (up to 4096 characters!)
- Pointer arithmetic support

1.3.4 Version 4 Release 2 (V4R2)

In Version 4, with new OS/400 releases starting to appear at much shorter intervals, the RPG IV compiler took advantage of the option to skip a release of the operating system. That is, there was no new RPG IV compiler for V4R1. The V3R7 RPG IV compiler worked with V4R1. However, at V4R2, the compiler made up for the wait by introducing another set of significant enhancements to the language.

The most significant enhancements in V4R2 surround the use of and replacement of numbered indicators in RPG IV. As of V4R2, it is now possible to write most RPG IV programs without the use of any numbered indicators (such as *IN10 through *IN99). Named indicators or the use of new built-in functions in place of indicators become the preferred method of communicating with display and printer files, as well as dealing with input/output operations. The following list contains the major enhancements for V4R2:

- Indicator data type (N)
- Ability to map DDS numbered indicators to named indicators
- Built-in functions for I/O and other types of condition handling (for example, %Error, %EOF, %Open, %Found)
- Support for varying length fields
- Compiler options can be embedded in source members on the H specification
- Enhanced external name support for imported or exported data items and procedures

1.3.5 Version 4 Release 4 (V4R4)

Version 4 Release 3, like V4R1, included no new RPG IV functions. At V4R4, many enhancements were made to add new functions to the compiler. The single biggest enhancement at this release was the ability to create thread safe RPG procedures.

Thread safety allows RPG procedures to be called by threaded applications, written in Java, for example. This will ultimately allow a tighter integration between Java applications and RPG applications on the AS/400 system. It also will allow for more possibilities of reusing RPG code within applications that are written in Java or other threaded environments, such as Domino or C++. The following list includes the major enhancements to RPG IV in V4R4:

- Thread safety enabled by option Thread(*Serialize)
- Ability to call RPG procedures in service programs from Java via Java Native Interface (JNI)
- New integer and Unicode data types
- EVALR operation code (Evaluate with right adjusted result)
- LEAVESR operation code (transfer control to ENDSR operation)
- New FOR loop syntax
- *Next option on Overlay keyword on D specification for Data Structure subfields
- New built-in function options: %XFOOT, %DIV, %REM
- Initialize a field to contain the user's ID
- Initialize externally defined data structures to the default values specified in DDS
- Ability to request matching source sequence numbers and program statements
- Ability to request I/O statements to represent only 1 step in a debug session

Special notice

The last two enhancements above (relating to source statement numbers and debugging I/O statements) were considered so important for customer satisfaction, they have been made available in earlier releases via PTF. To get this support in releases prior to V4R4, you must install the PTFs necessary for your release. You must also include the keywords `OPTION(*SRCSTMT *NODEBUGIO)` on the H specification in your programs. SEU may report a syntax error for this. If you ignore the error and save the source, the compiler will accept it. For specific PTF number information, see Appendix C, "PTFs for *SRCSTMT and *NODEBUGIO" on page 417.

1.3.6 The future for RPG IV

As the lists in the previous section indicate, there has been significant investment made in enhancing 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 and Java. Since Java support on the AS/400 system (and the IT industry in general) is rapidly growing, 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 (using the program call from the AS/400 Toolbox for Java or the JNI support in V4R4, for example), 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 we have 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.

1.3.7 A future for RPG programmers

As discussed in the previous section, 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 some information here 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.

1.3.8 A roadmap

It may be helpful to suggest a roadmap for programmers looking to move from pre-RPG IV environments to taking advantage of today’s RPG language.

1.3.8.1 Step 1: RPG IV

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) or

RPG36, now is the time to move forward. Learn RPG IV and move your applications to the new language as soon as possible.

As discussed in 1.4, “The relationship between the RPG IV language and ILE” on page 14, moving to RPG IV does not necessarily require the use of or the understanding of the Integrated Language Environment (ILE). Converting your code to RPG IV is simple, and there are many options of tools to help you do this. The productivity and maintainability gains alone make this step a good investment. And, it provides the required foundation for further steps.

To enhance your skills in RPG IV, there are many books, publications, classes offered by IBM and others, tutorials, and conferences that will provide help in understanding the differences between RPG IV and earlier versions of the RPG language. A few examples of these are listed in 1.6, “RPG IV sources on the Web” on page 19.

Once in RPG IV and throughout your journey, adopt a consistent style of coding in RPG IV. The style guide included in this book 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 will also help in making some basic style adjustments, such as the use of upper and lower case, the use of the D specification for stand-alone work fields, and the conversion, where possible, to the use of free-form operations, such as EVAL.

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.

1.3.8.2 Step 2: Modularization using ILE

After moving your applications to RPG IV, begin exploring the potential use of the Integrated Language Environment. 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 also provide the foundation for accessing most of the more modern system functions. It is also the foundation of integrating with Java.

This redbook will help you 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 a some other books, manuals, classes, articles, seminars, and conferences that can help you understand how to implement ILE in your applications. A small sampling of some of these is included in 1.6, “RPG IV sources on the Web” on page 19.

Look for opportunities to modularize your application code. Some examples of ways to do this are included in various chapters in this redbook. Modularization, when done well, usually results in higher productivity and reliability of the application logic. Many developers have also experienced improvements in application performance, primarily because of the ease of replicating performance improvements in modules used in multiple programs.

1.3.8.3 Step 3: Exploit database features

Another step to consider along this road is to look for opportunities to include database features to either replace or add reliability to parts of your application logic. This step may come before, after, or even simultaneously with the ILE implementation step mentioned above.

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, much 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.

1.3.8.4 Step 4: Modernize the user interface

Many, though not all, applications can be improved in usability or function by adding a graphical or Web-enabled user interface. For cases where a user interface that goes beyond the text-based capabilities of DDS display files is required, there are many options available. The type of user interface and the languages and tools used to create it depends on many factors, such as workstation type and variety, skills available among the developers, and whether universal access (via the Internet, for example) is required.

Java is a popular choice for user interface code and can be developed in many ways with a variety of development tools from which to choose. Some methods of interaction between Java and RPG applications are discussed in another redbook *Building AS/400 Applications with Java*, SG24-2163.

However, do not overlook other ways to Web-enable applications as well. Writing CGI bin programs in RPG, for example, may prove a simpler satisfactory solution for some application needs. See 5.6, “Writing CGI programs using RPG IV” on page 235, of this book for examples of Web-enabling RPG applications using CGI.

VisualAge for RPG is another potential vehicle for reusing your available RPG code and skills to develop graphical interfaces to your applications. Since the latest release of VisualAge for RPG generates Java source code, it can be used as a tool to write and maintain the interface logic in RPG, while deploying the application to a wide variety of user client choices. See Chapter 8, “VisualAge for RPG as a GUI for RPG applications” on page 395, for more information about using VisualAge for RPG to generate user interfaces to your RPG applications.

1.4 The relationship between the RPG IV language and ILE

A common source of confusion for those considering a move to RPG IV is about the relationship between the RPG IV language and the Integrated Language Environment (ILE). Because RPG IV is an ILE-enabled language, it is possible to create an application in RPG IV that fully exploits the capabilities of ILE. In addition, some of the features of the RPG IV language (subprocedures, for example) require the use of ILE. However, it is also quite possible to use the RPG IV language in a form that does not require the use of any ILE-specific system features.

Many customers that find using RPG IV in a "non-ILE" fashion is a good first step in an application conversion to take advantage of many of the new language features without requiring skill in the more complex features of ILE. This section describes the differences between using RPG IV in an ILE environment and in a non-ILE environment.

1.4.1 A brief introduction to the Integrated Language Environment (ILE)

The Integrated Language Environment was added to OS/400 in V2R3 as a new architecture for creating and running application programs. ILE was created primarily to more efficiently support applications that are written to a more modular fashion. There is a clear trend in the application development community in general to move away from writing compiled units of code measured in hundreds or thousands of lines of code. Instead, a more modular style is used, where compiled units of code are typically quite small.

Smaller units of code can allow for reduced complexity and easier reusability of application logic, which contribute to enhanced programmer productivity and to enhanced reliability of the application code. The Original Program Model (OPM), which is the architecture in use prior to the introduction of ILE, was quite good at supporting and managing code written in the earlier style of very large compilation units. However, it was less efficient at supporting more modular applications.

ILE provides the ability for the application developer to write and compile in small pieces of code and then statically bind those pieces together into program objects that are very efficient when calling between the pieces. In addition, at run time, the use of an ILE function, known as Activation Groups, allows for more efficient system management of the storage used by applications in each user's job. A more efficient exception handling model also enhances the performance and reliability of these new, more modular style applications. In Chapter 4, “An ILE guide for the RPG programmer” on page 61, there is a more detailed discussion of some of these ILE features.

1.4.2 Using RPG IV without ILE

There are many features of the RPG IV language that do not require the use of ILE system features. Even for applications that are not (yet) written in a modular style, there are many advantages to moving to the more readable and usable syntax of the RPG IV language. The RPG IV compiler supports creating program objects that are compatible with OPM programs.

There is a parameter on the Create Bound RPG Program (CRTBNDRPG) command called Default Activation Group (DFTACTGRP). The selection of the value *YES for this parameter creates a program object that behaves in a way compatible with OPM programs and can easily be mixed with OPM programs. The default Activation Group is the part of a user’s job where all OPM programs run. Typically, "true" ILE programs (that is, those programs created with DFTACTGRP(*NO)) run in named ILE Activation Groups. By specifying `DFTACTGRP(*YES)`, the programmer tells the compiler to create an OPM-compatible RPG IV program.

To begin taking advantage of the RPG IV language without any concern about a lack of skill in ILE concepts, simply ensure you always specify `DFTACTGRP(*YES)` when compiling your RPGLE source members. Then you can freely intermix your new RPG IV programs with your OPM programs.

However, when using this option (DFTACTGRP(*YES)), you will be unable to use any of the features of the RPG IV language that require the Integrated Language Environment (ILE). For example, you will not be able to use any form of a bound call in this type of program. This includes not only the use of the CALLB operation code, but also any use of RPG IV subprocedures.

As you are reading this book, it is important to note which features use (and require) ILE support. This will help you to understand how to compile the code you create and also which features may require a bit more planning effort for the use of ILE within the application.

1.5 IBM Certified Specialist AS/400 RPG programmer

Why should you become an IBM Certified AS/400 solutions expert? To demonstrate to your customers and colleagues that you have what it takes to design and develop superior custom solutions with IBM technologies. As a certified professional, you receive the following benefits:

- Industry recognition of your knowledge and proficiency with IBM AS/400 products and technologies.
- IBM Certification logo, certificate, wallet card, and lapel pin to enable you to identify your Certified status to colleagues or clients.

To learn more about IBM certification programs, go to the IBM certification Web site at: `http://www.ibm.com/certify`

1.6 RPG IV sources on the Web

Do you want to find out more about RPG IV on the Internet? Check out these resources.

1.6.1 General AS/400 resources from IBM

For general IBM AS/400 resources, consider these options:

- Both customers and IBM employees can find out about ITSO redbooks, redpieces, and CD-ROMs. A form for ordering books and CD-ROMs by fax or e-mail is also provided. For more information, see “How to get IBM Redbooks” on page 427. The redbooks Web site is located at:
  `http://www.redbooks.ibm.com/`
- IBM PartnerWorld for Developers AS/400 has the mission to help you build competitive solutions on the AS/400 system. Visit their site at:
  `http://www.as400.ibm.com/developer`
- AS/400 Information Starting Point (often called the InfoCenter due to its URL) is your gateway to AS/400 technical information and business solutions including the Information Center, Technical Studio and the Online Library of AS/400 reference material. You can find this information at:
  - `http://publib.boulder.ibm.com/pubs/html/as400/infocenter.html`
  - `http://www.as400.ibm.com/infocenter`

- Technical Studio has technical workshops and business solutions on a wide variety of AS/400 topics. Many of them feature RPG solutions. You can located Tech Studio at:
  - `http://publib.boulder.ibm.com/pubs/html/as400/techstudio.htm`
  - `http://www.as400.ibm.com/techstudio`
- For online documentation, go to: `http://publib.boulder.ibm.com/html/as400/onlinelib.htm` or `http://as400bks.rochester.ibm.com`

  Here, look for:
  - *ILE RPG for AS/400 Programmer's Guide*, SC09-2507
  - *ILE RPG for AS/400 Reference*, SC09-2508
- *IBM Integrated Language Environment (ILE) RPG for AS/400* is located at: `http://www.software.ibm.com/ad/as400/library/ilerpg44.html`
- VisualAge RPG and Code/400 is located at: `http://www.software.ibm.com/ad/varpg/`
- Refer to the course from IBM Learning Services: *Moving from RPG/400 to RPG IV,* S6126

1.6.2 Non-IBM general AS/400 resources

For non-IBM general AS/400 information, refer to these sources:

- A good AS/400 system specific search site can be found at: `http://www.search400.com`
- *Experience RPG IV Tutorial* by Rogers, by Masri & Santilli, can be found at: `http://www.advice.com`
- Robert Cozzi's RPG Web site is located at: `http://www.rpgiv.com`
- *The Modern RPG IV Language*, by Robert Cozzi, can be found at: `http://www.mc-store.com`
- The 400 Group, which has an ILE RPG IV focus area is located at: `http://www.the400group.com`
- Midrange: `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 AS/400.
- *NEWS/400* has active forums (like a news group) on RPG at: `http://www.news400.com/navbar/Glance-Forums.html`
- The AS/400 user group called COMMON is located on the Web at: `http://www.common.org`

  This is the USA Web site. 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: `http://www.comeur.org/f_events.htm`
- AS/400 Manager's online resource is available at: `http://www.hotlink400.com`
- An eclectic collection of AS/400 code examples from *NEWS/400* Tips and Techniques Community is on the Web at: `http://www.tnt400.com/codepage400.htm`

- *RPG IV Jump Start*, by Bryan Meyers, can be found at:
  `http://www.news400store.com`
- News groups:
  - `comp.sys.ibm.as400.misc`
  - `news.software.ibm.com`
  - `ibm.software.code400`
  - `ibm.software.varpg`

1.6.3 AS/400 magazines and books on the Web

The following publications and publishers can be found on the Internet:

- IBM's *AS/400 Magazine*: `http://www.as400magazine.com`
- *NEWS/400*: `http://www.news400.com`
- *Midrange Computing*: `http://www.midrangecomputing.com`
- *Midrange Systems*: `http://www.midrangesystems.com`
- Duke Communications: `http://www.dukepress.com`
- Advice Press: `http://www.advice.com`

Chapter 2. Programming RPG IV with style

This section of the book takes as its starting point "The Essential RPG IV Style Guide" by Bryan Meyers, CCP, a senior technical editor for *NEWS/400*.

This article was first published in the June 1998 edition of *NEWS/400*. As Mr. Meyers noted in the original article, the evolution of this style guide was a collaborative effort. It emerged over a period of months out of discussions that took place in *The RPG IV Style Forum* on the *NEWS/400* Web site at:
`http://www.news400.com`

Not content with merely “preaching”, we also attempted to apply its rules to the coding samples provided in this redbook. As any style guide is just a guide, we took a small number of liberties with the source code samples provided with this redbook since it was not practical in all cases to convert the code to a single style since the code was gathered from a wide range of sources.

To preserve the original content of the work collected by Bryan Meyers, we follow this section with 2.2, “Comments and extensions to the style guide” on page 27. This section offers our own commentary to clarify or expand on some of the style suggestions raised.

2.1 The essential RPG IV style guide

Here is a copy of the "The Essential RPG IV Style Guide" article from the June 1998 edition of *NEWS/400*. We felt that it summarized the intent of this section so well that it warranted reprinting the article here in its entirety.

2.1.1 Comments

Good programming style can serve a documentary purpose in helping others understand the source code. If you practice good code-construction techniques, you’ll find that “less is more” when it comes to commenting the source. Too many comments are as bad as too few.

2.1.1.1 Use comments to clarify, not echo, your code

Comments that merely repeat the code add to a program’s bulk, but not to its value. In general, you should use comments for just three purposes:

- To provide a brief program or procedure summary
- To give a title to a subroutine, procedure, or other section of code
- To explain a technique that isn’t readily apparent by reading the source

2.1.1.2 Always include a brief summary at the beginning of a program or procedure

This prologue should include the following information:

- A program or procedure title
- A brief description of the program’s or procedure’s purpose
- A chronology of changes that includes the date, programmer name, and purpose of each change
- A summary of indicator usage
- A description of the procedure interface (the return value and parameters)
- An example of how to call the procedure

2.1.1.3 Use consistent ‘marker line’ comments to divide major sections of code

For example, you should definitely section off with lines of dashes or asterisks the declarations, the main procedure, each subroutine, and any subprocedures. Identify each section for easy reference.

2.1.1.4 Use blank lines to group related source lines and make them stand out

In general, you should use completely blank lines instead of blank comment lines to group lines of code, unless you’re building a block of comments. Use only one blank line, though; multiple consecutive blank lines make your program hard to read.

Avoid right-hand “end-line” comments in columns 81 to 100. Right-hand comments tend simply to echo the code, can be lost during program maintenance, and can easily become “out of sync” with the line they comment. If a source line is important enough to warrant a comment, it’s important enough to warrant a comment on a separate line. If the comment merely repeats the code, eliminate it entirely.

2.1.2 Declarations

With RPG IV, we finally have an area of the program source in which to declare all variables and constants associated with the program. The D-specs organize all your declarations in one place.

2.1.2.1 Declare all variables within D-specs

Except for key lists and parameter lists, do not declare variables in C-specs — not even using *LIKE DEFN. Define key lists and parameter lists in the first C-specs of the program, before any executable calculations.

2.1.2.2 Declare a literal as a named constant in the D-specs

This practice helps document your code and makes it easier to maintain. One obvious exception to this rule is the allowable use of 0 and 1 when they make perfect sense in the context of a statement. For example, if you’re going to initialize an accumulator field or increment a counter, it’s fine to use a hard-coded 0 or 1 in the source.

2.1.2.3 Indent data item names for readability and document data structures

Unlike many other RPG entries, the name of a defined item need not be left-justified in the D-specs; take advantage of this feature to help document your code:

```
D ErrMsgDSDS      DS
D   ErrPrefix                    3
D   ErrMsgID                     4
D   ErrMajor                     2    OVERLAY(ErrMsgID:1)
D   ErrMinor                     2    OVERLAY(ErrMsgID:3)
```

2.1.2.4 Use length notation instead of positional notation in data structure declarations

D-specs let you code fields either with specific from and to positions or simply with the length of the field. To avoid confusion and to better document the field, use length notation consistently. For example, code:

```
D RtnCode      DS
D  PackedNbr                 15P 5
instead of
D RtnCode      DSD  PackedNbr            1     8P 5
```

2.1.2.5 Use positional notation only when the actual position in a data structure is important

For example, when coding the program status data structure, the file information data structure, or the return data structure from an API, you'd use positional notation if your program ignores certain positions leading up to a field or between fields. Using positional notation is preferable to using unnecessary "filler" variables with length notation:

```
D APIRtn       DS
D  PackedNbr            145   152P 5
```

In this example, to better document the variable, consider overlaying the positionally declared variable with another variable declared with length notation:

```
D APIRtn       DS
D  Pos145               145   152
D   PackNbr                    15P 5  OVERLAY(Pos145)
```

2.1.2.6 When defining overlapping fields, use the OVERLAY keyword instead of positional notation

Keyword OVERLAY explicitly ties the declaration of a "child" variable to that of its "parent." Not only does OVERLAY document this relationship, but if the parent moves elsewhere within the program code, the child will follow.

2.1.2.7 If your program uses compile-time arrays, use the **CTDATA form to identify the compile-time data

This form effectively documents the identity of the compile-time data, tying the data at the end of the program to the array declaration in the D-specs. The **CTDATA syntax also helps you avoid errors by eliminating the need to code compile-time data in the same order in which you declare multiple arrays.

2.1.3 Naming conventions

Perhaps the most important aspect of programming style deals with the names you give to data items (for example variables and named constants) and routines.

2.1.3.1 When naming an item, be sure the name fully and accurately describes the item

The name should be unambiguous, easy to read, and obvious. Although you should exploit RPG IV's allowance for long names, do not make your names too long to be useful. Name lengths of 10 to 14 characters are usually sufficient, and longer names may not be practical in many specifications. When naming a data item, describe the item; when naming a subroutine or procedure, use a verb/object syntax (similar to a CL command) to describe the process. Maintain a dictionary of names, verbs, and objects, and use the dictionary to standardize your naming conventions.

2.1.3.2 When coding an RPG symbolic name, use mixed case to clarify the named item's meaning and use

RPG IV lets you type your source code in upper and lowercase characters. Use this feature to clarify named data. For RPG-reserved words and operations, use all uppercase characters.

2.1.3.3 Avoid using special characters (for example, @, #, $) when naming items

Although RPG IV allows an underscore (_) within a name, you can easily avoid using this "noise" character if you use mixed case intelligently.

2.1.4 Indicators

Historically, indicators have been an identifying characteristic of the RPG syntax, but with RPG IV they are fast becoming relics of an earlier era. To be sure, some operations still require indicators, and indicators remain the only practical means of communicating conditions to DDS-defined displays. But reducing a program's use of indicators may well be the single most important thing you can do to improve the program's readability.

2.1.4.1 Use indicators as sparingly as possible; go out of your way to eliminate them

In general, the only indicators present in a program should be resulting indicators for opcodes that absolutely require them (for example, CHAIN before V4R2) or indicators used to communicate conditions such as display attributes to DDS-defined display files.

Whenever possible, use Built-in Functions (BIFs) instead of indicators

As of V4R2, you can indicate file exception conditions with error-handling BIFs (for example, %EOF, %ERROR, %FOUND) and an E operation extender to avoid using indicators.

2.1.4.2 If you must use indicators, name them

V4R2 supports a Boolean data type (N) that serves the same purpose as an indicator. You can use the INDDS keyword with a display-file specification to associate a data structure with the indicators for a display or printer file; you can then assign meaningful names to the indicators.

Use an array-overlay technique to name indicators before V4R2

Using RPG IV's pointer support, you can overlay the *IN internal indicator array with a data structure. Then you can specify meaningful subfield names for the indicators. This technique lessens your program's dependence on numeric indicators. For example:

```
D IndicatorPtr                    *   INZ(%ADDR(*IN))
D               DS                    BASED(IndicatorPtr)
D F03Key                  3     3
D F05Key                  5     5
D CustNotFnd             50    50
D SflClr                 91    91
D SflDsp                 92    92
D SflDspCtl              93    93
```

```
C                   IF          F03Key = *ON
C                   EVAL        *INLR = *ON
C                   RETURN
C                   ENDIF
```

2.1.4.3 Use the EVAL opcode with *Inxx and *ON or *OFF to set the state of indicators

Do not use SETON or SETOFF, and never use MOVEA to manipulate multiple indicators at once.

2.1.4.4 Use indicators only in close proximity to the point where your program sets their condition

For example, it's bad practice to have indicator 71 detect end-of-file in a READ operation and not reference *IN71 until several pages later. If it's not possible to keep the related actions (setting and testing the indicator) together, move the indicator value to a meaningful variable instead.

2.1.4.5 Do not use conditioning indicators — ever

If a program must conditionally execute or avoid a block of source, explicitly code the condition with a structured comparison opcode, such as IF. If you're working with old S/36 code, get rid of the blocks of conditioning indicators in the source.

2.1.4.6 Include a description of any indicators you use

It's especially important to document indicators whose purpose isn't obvious by reading the program, such as indicators used to communicate with display or printer files or the U1-U8 external indicators, if you must use them.

2.1.5 Structured programming techniques

Give those who follow you a fighting chance to understand how your program works by implementing structured programming techniques at all times.

2.1.5.1 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 exits from loops.

2.1.5.2 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 those alternatives more readable. In general, if an opcode offers a free-format alternative, use it.

2.1.5.3 Perform multipath comparisons with SELECT, WHEN, OTHER, ENDSL

Deeply nested IFxx/ELSE/ENDIF code blocks are hard to read and result in an unwieldy accumulation of ENDIFs at the end of the group. Do not use the obsolete CASxx opcode; instead, use the more versatile SELECT/WHEN/OTHER/ENDSL construction.

2.1.5.4 Always qualify END opcodes
Use ENDIF, ENDDO, ENDSL, or ENDCS as applicable. This practice can be a great help in deciphering complex blocks of source.

2.1.5.5 Avoid programming tricks and hidden code
Such maneuvers aren't so clever to someone who doesn't know the trick. If you think you must add comments to explain how a block of code works, consider rewriting the code to clarify its purpose. Use of the obscure "bit-twiddling" opcodes (BITON, BITOFF, MxxZO, TESTB, TESTZ) may be a sign that your source needs updating.

2.1.6 Modular programming techniques
The RPG IV syntax, along with the AS/400's Integrated Language Environment (ILE), encourages a modular approach to application programming. Modularity offers a way to organize an application, facilitate program maintenance, hide complex logic, and efficiently reuse code wherever it applies.

2.1.6.1 Use RPG IV's prototyping capabilities to define parameters and procedure interfaces
Prototypes (PR definitions) offer many advantages when you're passing data between modules and programs. For example, they avoid runtime errors by giving the compiler the ability to check the data type and number of parameters. Prototypes also let you code literals and expressions as parameters, declare parameter lists (even the *ENTRY PLIST) in the D-specs, and pass parameters by value and by read-only reference, as well as by reference.

2.1.6.2 Store prototypes in /COPY members
For each module, code a /COPY member containing the procedure prototype for each exported procedure in that module. Then, include a reference to that /COPY module in each module that refers to the procedures in the called module. This practice saves you from typing the prototypes each time you need them and reduces errors.

Note

Include constant declarations for a module in the same /COPY member as the prototypes for that module.

If you then reference the /COPY member in any module that refers to the called module, you've effectively "globalized" the declaration of those constants.

2.1.6.3 Use IMPORT and EXPORT only for global data items
The IMPORT and EXPORT keywords let you share data among the procedures in a program without explicitly passing the data as parameters — in other words, they provide a "hidden interface" between procedures. Limit use of these keywords to data items that are truly global in the program — usually values that are set once and then never changed.

2.1.7 Character string manipulation

IBM has greatly enhanced RPG IV's ability to easily manipulate character strings. Many of the tricks you had to use with earlier versions of RPG are now obsolete. Modernize your source by exploiting these new features.

2.1.7.1 Use a named constant to declare a string constant instead of storing it in an array or table

Declaring a string (such as a CL command string) as a named constant lets you refer to it directly instead of forcing you to refer to the string through its array name and index. Use a named constant to declare any value that you do not expect to change during program execution.

2.1.7.2 Avoid using arrays and data structures to manipulate character strings and text

Use the new string manipulation opcodes and/or built-in functions instead.

2.1.7.3 Use EVAL's free-format assignment expressions whenever possible for string manipulation

When used with character strings, EVAL is usually equivalent to a MOVEL(P) opcode. Use MOVE and MOVEL only when you do not want the result to be padded with blanks.

2.1.8 Avoid obsolescence

RPG is an old language. After 30 years, many of its original, obsolete features are still available. *Do not use them*.

2.1.8.1 Do not sequence program line numbers in columns 1 through 5

Chances are you'll never again drop that deck of punched cards, so the program sequence area is unnecessary. In RPG IV, the columns are commentary only. You may use them to identify changed lines in a program or structured indentation levels, but be aware that these columns may be subject to the same hazards as right-hand comments.

2.1.8.2 Avoid program-described files

Instead, use externally defined files whenever possible.

2.1.8.3 If an opcode offers a free-format syntax, use it instead of the fixed-format version

Opcodes to avoid include CABxx, CASxx, CAT, DOUxx, DOWxx, IFxx, and WHxx.

2.1.8.4 If a BIF offers the same function as an opcode, use the BIF instead of the opcode

With some opcodes, you can substitute a built-in function for the opcode and use the function within an expression. At V4R1, the SCAN and SUBST opcodes have virtually equivalent built-in functions, %SCAN and %SUBST. In addition, you can usually substitute the concatenation operator (+) in combination with the %TRIMx BIFs in place of the CAT opcode. The free-format versions are preferable if they offer the same functionality as the opcodes.

2.1.8.5 Shun obsolete opcodes

In addition to the opcodes mentioned earlier (style guidelines 5.2 and 5.3), some opcodes are no longer supported or have better alternatives:

- **CALL, CALLB**

  The prototyped calls (CALLP or a function call) are just as efficient as CALL and CALLB and offer the advantages of prototyping and parameter passing by value. Neither CALL nor CALLB can accept a return value from a procedure.

- **DEBUG**

  With OS/400's advanced debugging facilities, this opcode is no longer supported.

- **DSPLY**

  You should use display file I/O to display information or to acquire input.

- **FREE**

  This opcode is no longer supported.

- **PARM, PLIST**

  If you use prototyped calls, these opcodes are no longer necessary.

2.1.9 Miscellaneous guidelines

Here's an assortment of other style guidelines that can help you improve your RPG IV code.

2.1.9.1 In all specifications that support keywords, observe a one-keyword-per-line limit

Instead of spreading multiple keywords and values across the entire specification, your program will be easier to read and let you more easily add or delete specifications if you limit each line to one keyword, or at least to closely related keywords (for example, DATFMT and TIMFMT).

Begin all H-spec keywords in column 8, leaving column 7 blank

Separating the keyword from the required H in column 6 improves readability.

2.1.9.2 Relegate mysterious code to a well-documented, well-named procedure

Despite your best efforts, on extremely rare occasions, you simply will not be able to make the meaning of a chunk of code clear without extensive comments. By separating such heavily documented, well-tested code into a procedure, you'll save future maintenance programmers the trouble of deciphering and dealing with the code unnecessarily.

2.1.10 Recommendations

Sometimes good style and efficient runtime performance do not mix. Wherever you face a conflict between the two, choose good style. Hard-to-read programs are hard to debug, hard to maintain, and hard to get right. Program correctness must always win out over speed. Keep in mind these admonitions from Brian Kernighan and P.J. Plauger's *The Elements of Programming Style*:

- Make it right before you make it faster.
- Keep it right when you make it faster.
- Make it clear before you make it faster.
- Do not sacrifice clarity for small gains in efficiency.

2.2 Comments and extensions to the style guide

We do not expect you to agree with all the points made in the style guide, but it is an excellent foundation. Every installation is different, and we encourage you to use these guidelines as a starting point for your own.

Look at the standards in use in your own installation. You may not have any formal documentation, but your programs will undoubtedly express some style, even if it is best described as "early 70s."

As we reviewed the style guide, we made a few additions, clarifications and modifications of our own. You may find them useful in establishing your own "style".

- *Section 2.1.2.3, "Indent data item names for readability and document data structures" on page 20*

  When coding D specs, the RPG compiler allows you to start the name of the data item in position 7. *Resist the temptation*! Always leave at least one space between the D and the first character of the name. You will be amazed at how much more readable it makes your code. Compare the two samples that follow if you have any doubts:

  ```
  DWorkNum          S              7  0
  DWorkDay          S              1  0
  DDayName          S              9
  DWorkDate         S               D   DatFmt(*ISO)
  ```

  We think you'll agree that this second version is a lot more readable, and more readable means more maintainable:

  ```
  D WorkNum         S              7  0
  D WorkDay         S              1  0
  D DayName         S              9
  D WorkDate        S               D   DatFmt(*ISO)
  ```

- *Section 2.1.2.6, "When defining overlapping fields, use the OVERLAY keyword instead of positional notation" on page 21*

  Those of you who have been using RPG IV for a while may not have noticed that additional function has been added to the OVERLAY keyword. It is now possible to specify the name of a data structure as the "parent" as you can see in the following example:

  ```
  D DayData         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')

  D DayName                        9A   Dim(7) Overlay(DayData)
  ```

  Also in V4R4, the keyword *NEXT was added as an option so that a hard-coded offset does not need to be used.

- *Section 2.1.3.2, “When coding an RPG symbolic name, use mixed case to clarify the named item’s meaning and use” on page 22*

  While we agree wholeheartedly with using mixed case, we find that many RPG programmers seem to be allergic to using the Shift key. We can only plead with them to at least type all of their code in lower case rather than all in upper case.

  One example of a mixed-case naming convention is the "Camel Notation" (for example: ActStsCde for account status code). The naming convention refers to the "humps" in the variable names.

  This guideline also suggests that you use all upper case for RPG reserved words and operations. We feel that this is largely a matter of taste and have chosen to use mixed-case in our examples.

- *Section 2.1.3.3, “Avoid using special characters (for example, @, #, $) when naming items” on page 22*

  We have deliberately broken this rule in one instance—in the naming of pointers. In this case, we use the "@" (commercial at) symbol at the end of the field name. For example, a pointer field that is used to supply the address of the data structure AddrDetail would be named AddrDetail@.

  Avoiding using special characters takes on new meaning when you take into account international languages. The invariant character set is important, because this is the common part of characters used in all codepages. Interesting to see that the example of the "@" used have different X'..' values in a lot of codepages. This can be a significant problem if you start re-using modules compiled in another country (with different codepage) than the country that you will bind into a program.

  An example of how to avoid the "@" symbol to indicate the variable is a pointer is to use the "Hungarian Notation." Here, the first character of the variable name indicates the data type of the variable, for example, pCustNbr or cActStsCde.

- *Section 2.1.4.2, “If you must use indicators, name them” on page 22*

  Some readers, particularly those who do not have access to V4R2 named indicators, may prefer the following technique as an alternative to the array-overlay method of naming indicators mentioned in the style guide. The basic idea here is to take advantage of the fact that RPG allows indicators to be treated as an array. By using a named constant with a value of the required indicator number, the constant can be used to name it, for example:

```
D F03Key          C                   3
D F05Key          C                   5
D CustNotFnd      C                   50

C     CustKey       Chain     CustMaster                         50
C                   If        *In(CustNotFnd) = *On
```

- *Section 2.1.4.3, “Use the EVAL opcode with *Inxx and *ON or *OFF to set the state of indicators” on page 23*

  We recommend that you always use *On and *Off rather than the obscure "1" and "0". It is also worth noting that in RPG IV, an indicator's status can be tested directly. For example, the following lines of code are equivalent:

```
C                   If        *In25 = *On and *In30 = *Off

C                   If        *In25 and Not *In30
```

- *Section 2.1.5.5, "Avoid programming tricks and hidden code" on page 24*

  If you have absolutely no choice but to "bit-twiddle", consider isolating the code in a subprocedure or subroutine. That way it is less likely to be "broken" during maintenance by a programmer who does not understand its intent. The style guide also suggests this tactic in 2.1.9, "Miscellaneous guidelines" on page 26.

- *Section 2.1.6.2, "Store prototypes in /COPY members" on page 24*

  Some programmers prefer to group all of the prototypes for a particular Service Program into a single /COPY member, rather than the "one per module" approach. It is also acceptable to group prototypes based on the subsystem in which they are used.

  The important point to note here is that prototypes do not cause any storage to be allocated. This frees you to group your prototypes in any manner that suits your installation since it doesn't matter how many "unused" prototypes appear in a program.

  If you include Constants in the /Copy member, watch out for situations where you may have the same named constant defined in multiple places. In this case, you may find it necessary to use RPG IV's ability to nest /Copy statements and incorporate the constants in this way.

  You may also want to take advantage of V4R2 conditional compilation directives to ensure that a /COPY member is not included more than once.

- *Section 2.1.7.2, "Avoid using arrays and data structures to manipulate character strings and text" on page 25*

  This is even more significant if you are using V4R2 or later releases. On these releases, you should look at using variable length string support when manipulating strings.

Chapter 3. Subprocedures

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

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. By this, we are referring to programs that are created by the Create Program (CRTPGM) command or by Create Bound RPG Program (CRTBNDRPG) command with the Default activation group (DFTACTGRP) parameter set to *NO.

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 *ILE RPG for AS/400 Reference*, SC09-2508.

3.1.3 Built-in functions

Subprocedures can optionally 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 free-form C 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 variable 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

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 may have defined for use within a subroutine. Such variables are accessible from *anywhere* in the entire source file 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 a parameter passing on the AS/400 system, where parameters are passed by reference. By this, we mean 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.

  The RPG compiler insists that all subprocedures be prototyped (see 3.3.1.1, “Prototypes” on page 35). This allows the parameters passed to a 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 free-form C 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, we have used the word "spec" to refer to specification. This applies to all occurrences of the 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 in 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).

A note on recursion

Each recursive call causes a new invocation of the procedure to be placed on the call stack. The new invocation has new storage for all data items in automatic storage, and that storage is unavailable to other invocations because it is local. A data item that is defined in a subprocedure uses automatic storage unless the STATIC keyword is specified for the definition.

The automatic storage that is associated with earlier invocations is unaffected by later invocations. All invocations share the same static storage, so later invocations can affect the value held by a variable in static storage.

Recursion can be a powerful programming technique when properly understood.

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:

- After the main source section if one exists
- Following an H spec containing the *NOMAIN* keyword if there is no main section

  Note: There may be other specs between the H spec and the beginning of the subprocedure, for example, F specs to define files to be used by the subprocedures.

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

- Prototype (see 3.3.1.1, “Prototypes” on page 35, 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 following simple subprocedure, which receives two integers passed as parameters and returns the sum of the two.

Note: The markers **1** through **6** are defined immediately after this source code example.

```
 * PROCEXAM from QRPGLESRCS in RPGISCOOL

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

P AFunction       B                                         1

D AFunction       PI            10P 0                       2
D  AnInputParm1                  5P 0
D  AnInputParm2                  5P 0

D AReturnValue    S             10P 0                       3

C                   Eval      AReturnValue = AnInputParm1   4
C                                           + AnInputParm2
C                   Dsply                  AReturnValue

C                   Return    AReturnValue                  5

P                 E                                         6
```

1 The Begin procedure spec (B in position 24 of a procedure spec) supplies the name of the subprocedure. Its presence also signals to the compiler not to flag the subsequent D and C 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 our example, there are two parameters. The end of the list is signalled by the appearance of the standalone field AReturnValue.

The procedure-interface definition is optional if the subprocedure does not return a value and does not have any parameters passed to it. Please refer to 3.3.2, "Procedure-interface definitions" on page 36.

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 may refer to both local and global definitions, though 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 our example by coding:

```
C                   Return    AnInputParm1 + AnInputParm2
```

6 The End procedure spec (E in position 24 of a procedure spec). The name of the subprocedure can be repeated here, but is not required.

3.3.1.1 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 this:

```
 * PROCEXAMPR from QRPGLESRCS in RPGISCOOL

D AFunction       PR            10P 0
D  Packed5                       5P 0
D  Packed5                       5P 0
```

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. We recommend 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 the 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 needs also to be included when compiling the module in which the subprocedure is located, and when compiling any module that wants to use the subprocedure.

Language interoperability

This section concentrates on writing and using only 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 OS/400 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.

The concept of prototyping the C function library is further explained in 5.1, “Exploiting the C function library: A case study” on page 119.

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. In practice, most programmers tend to place them at the beginning of the D 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 D specs providing that it is preceded by its prototype. Again, it is a good idea to have these as the first two items in the D specs.

3.3.3 Order of coding the source elements

As we noted earlier, 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 H, F, D, I, C, and O 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 1 illustrates the layout of a complete RPG source containing multiple subprocedures.

Note: The markers 1 through 5 are defined immediately after this figure.

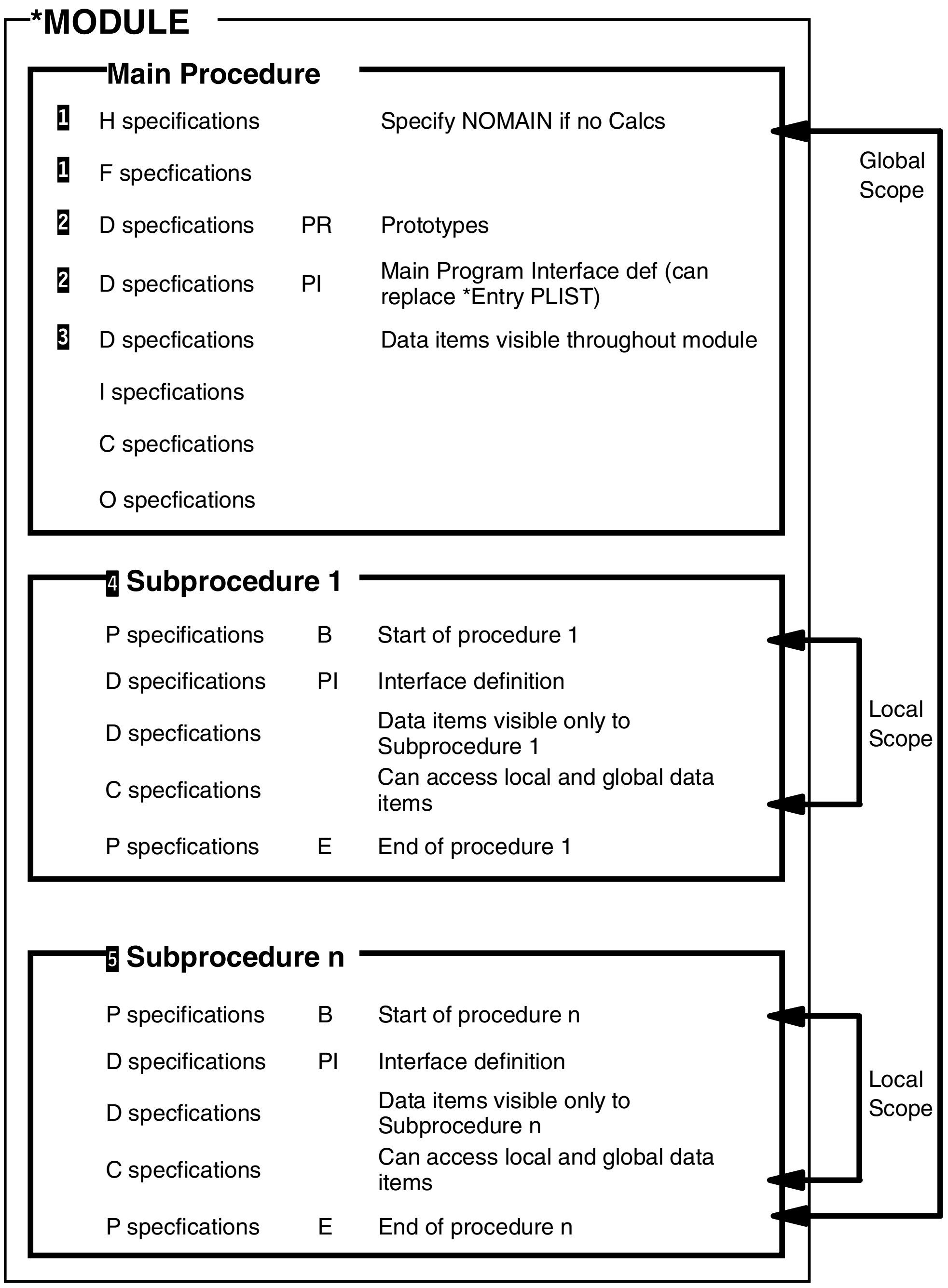


Figure 1. Order of coding the source elements

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

Note that the F specs always go at the beginning of the member, just after the H spec. These files can be used, either by the mainline code and/or by the subprocedures. Even if there is no main line logic, the F specs must always be defined at the beginning of the module. By definition then, all fields defined in files are global in nature.

2 The first PI line in the program serves as a replacement for the *ENTRY PLIST. This is optional.

3 The D and I specs that follow are for data items in the mainline, which are global. They can be accessed from both mainline logic and any subprocedures in this module.

4 Following the O specs for the mainline code is the beginning P spec for the first subprocedure. It is followed by the PI (procedure interface). The D and C specs for this subprocedure are next. The final line is the ending P spec.

5 Any other subprocedures would follow, each with its own set of P, D, and C specs.

The procedure-interface definition may be placed anywhere within the definition specs. However, we strongly recommend that you code it immediately following the P 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 of the calculation specs.

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 we discuss later (see 3.6.1, “The power of prototyping” on page 50). For these reasons, we only discuss the use of prototyped subprocedure calls in this section.

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.

There are two ways to make a free-form call:

- If there is no return value, use the CALLP operation.
- If there is a return value, place the prototyped procedure within an expression, for example, using the EVAL instruction. If you do not the want to use the returned value, you can use CALLP, in which case, 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 56

Here are some examples of using the subprocedure AFunction which we introduced earlier in this section (see 3.3.1, “Subprocedure definition” on page 33). 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.

```
 * Using the free-form function call with the Eval op-code
C                   Eval      TheResult = AFunction(AParm1 : AParm2)

 * Using the free-form function call with the If op-code
C                   If        AFunction(Parm1 : Parm2) = 1

 * Using the free-form function call with the Dou op-code
C                   Dou       AFunction(Parm1 : Parm2) = 1
```

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

```
 * Using the Call a Prototyped Procedure (CallP) op-code
C                   CallP     AFunction(AParm1 : AParm2)
```

3.4 Moving from subroutines to subprocedures

Existing subroutines often make good candidates for subprocedures. In this section, we take a subroutine and convert it into a subprocedure.

3.4.1 Why use subprocedures

As we noted in 3.2, “Advantages of using subprocedures” on page 32, 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

This program example 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 see the same code re-written with subprocedures.

```
  * filename DATESUBR from SUBPROCSRC in RPGISCOOL
  * This routine uses Sunday as day 7 - any date that represents
  *   a Sunday will work
 D AnySunday       C                   D'1999-06-13'

 D WorkNum         S              7  0
 D WorkDay         S              1  0
 D DayName         S              9
 D WorkDate        S               D   DatFmt(*ISO)

  * Days of the week name table - note no field names are required
 D NameData        DS
 D                                9    Inz('Monday')
 D                                9    Inz('Tuesday')
 D                                9    Inz('Wednesday')
 D                                9    Inz('Thursday')
 D                                9    Inz('Friday')
 D                                9    Inz('Saturday')
 D                                9    Inz('Sunday')
  * Define the array as an overlay of the DS name
 D  Name                          9    Dim(7) Overlay(NameData)

  * Program parameters
 C     *Entry        PList
 C                   Parm                    WorkDate

  * Call DayOfWeek subroutine to initialize field Workday
 C                   Exsr      DayOfWeek

  * Using Workday, initialize DayName from name table
 C                   Eval      DayName = Name(WorkDay)

  * Display resulting name
 C     DayName       Dsply

  * Terminate Program
 C                   Eval      *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)

 C     DayOfWeek     Begsr

 C     WorkDate      Subdur    AnySunday     WorkNum:*D
 C     WorkNum       Div       7             WorkNum
 C                   Mvr                     WorkDay

  * Testing for < 1 allows for the situation where the input date
  *   is earlier than the base date (AnySunday)

 C                   If        WorkDay < 1
 C                   Add       7             WorkDay
 C                   Endif

 C                   Endsr
```

Try it yourself

You can try this example by compiling the code from this section on your AS/400 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. They are covered later in the sections that are indicated:

- Use a procedure interface for the Main procedure's parameters. See 3.5.2, "Replacing the *ENTRY PLIST" on page 45.
- Use other subprocedures in your subprocedure. See 3.5.3, "Subprocedures using subprocedures" on page 46.
- Place your subprocedures in a Service Program. See 3.5.4, "Using an ILE Service Program" on page 47.

Note: In this section, the markers 1 through 5 next to the code snippets in the text correspond to the same numbers in the full program listing found in 3.4.4, "DATEMAIN1 subprocedure example" on page 43.

Doing it with style

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

3.4.3.1 Step 1: Returning a value

This first step focus 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 will be returned to the caller. This can be a simple value:

   ```
   C                   If        WorkDay < 1
   C                   Add       7             WorkDay
   C                   Endif
    * Returning result to the calling procedure                  1
   C                   Return    WorkDay
   ```

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

   ```
    * Returning result to the calling procedure                  1
   C                   If        WorkDay < 1
   C                   Return    WorkDay + 7
   C                   Else
   C                   Return    WorkDay
   C                   Endif
   ```

3.4.3.2 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 P-specs (Begin and End).

   Subprocedures begin and end with P specs. The beginning P spec names the subprocedure and has a similar layout to the D spec. The B (for Begin) and the E (for End) appear in position 24. Note that the ending P spec doesn't require that the name of the procedure be present.

   ```
    * SubProcedure definition: DayOfWeek                        2
   P DayOfWeek       B
   ...
   P                 E
   ```

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 our example, this is a single digit numeric field with no decimal places.

   Subsequent lines define the parameters passed to or from the subprocedure. Our 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 the start of a new data structure, stand-alone field, constant, or prototype. In our example, the parameter list is terminated by the arrival of the constant AnySunday.

   ```
    * SubProcedure definition: DayOfWeek                         3
   D DayOfWeek       PI             1  0
   ```

```
D  WorkDate                      D

D AnySunday       C                   D'1999-06-13'
```

For the purpose of this example, we moved those field definitions and constants, which are used only by the subprocedure, into the subprocedure itself. These local data items will 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 those 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. See 3.6.2.1, “Passing by reference” on page 54.
- Use ILE’s ability to share data items between modules via the IMPORT or EXPORT keyword. See Chapter 4, “An ILE guide for the RPG programmer” on page 61.

3.4.3.3 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
 D DayOfWeek       PR              1  0
 D  InputDate                       D   Datfmt(*ISO)
```

3.4.3.4 Step 4: Calling the subprocedure

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

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

```
D DayNo           S              1  0
....
 * Call DayOfWeek subprocedure, passing WorkDate, will return DayNo
C                    Eval      DayNo = DayOfWeek(WorkDate)
 * Using DayNoas an index to table Name to derive DayName value
C                    Eval      DayName = Name(DayNo)
```

However, there’s a better way. Remember that subprocedures can appear anywhere in an expression where a variable of the same type can be used.

Here's the same example with a "cleaner" programming style:

```
 * Call DayofWeek subprocedure, passing WorkDate, using return      5
 *  value to derive DayName value.

C                    Eval      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

Here is 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 39. As described earlier, the following steps enhance this code to take advantage of the power of using subprocedures efficiently in RPG IV.

```
 * DATEMAIN1 from SUBPROCSRC in RPGISCOOL

 * Prototype for subprocedure DayOfWeek                          4
D DayOfWeek       PR             1  0
D  InputDate                      D   Datfmt(*ISO)

 * Days of the week name table - note no field names are required
D NameData        DS
D                                9    Inz('Monday')
D                                9    Inz('Tuesday')
D                                9    Inz('Wednesday')
D                                9    Inz('Thursday')
D                                9    Inz('Friday')
D                                9    Inz('Saturday')
D                                9    Inz('Sunday')
 * Define the array as an overlay of the DS name
D  Name                          9    Dim(7) Overlay(NameData)

D DayName         S              9
D WorkDate        S               D   DatFmt(*ISO)

 * Program input parameter
C     *Entry        PList
C                   Parm                    WorkDate

 * Using DayofWeek, initialize DayName with table Name          5
C                   Eval      DayName = Name(DayOfWeek(WorkDate))

 * displaying result
C     DayName       Dsply

 * Terminate Program
C                   Eval      *InLR = *On

 * 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)                              2

P DayOfWeek       B

 * procedure interface definition                               3
D DayOfWeek       PI             1  0
D  WorkDate                       D

D AnySunday       C                   D'1999-06-13'

D WorkNum         S              7  0
D WorkDay         S              1  0

C     WorkDate      Subdur    AnySunday     WorkNum:*D
C     WorkNum       Div       7             WorkNum
C                   Mvr                     WorkDay

 * Returning result to the calling procedure                    1
```

```
C                   If        WorkDay < 1
C                   Return    WorkDay + 7
C                   Else
C                   Return    WorkDay
C                   Endif

 * Procedure definition end marker                              2
P                 E
```

Try it yourself

You can try this example by compiling the code from this section on your AS/400 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

In this section, we enhance the previous example by:

- Using /COPY members for prototypes
- Replacing the main procedure's *ENTRY PLIST with a Procedure Interface
- Using a subprocedure from within our subprocedure
- Creating a Service Program from our subprocedures

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's no performance penalty for unused prototypes.

Please note that, when we say /COPY, we do *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 following examples list a prototype in their specific members and include them in the main source section when required. The /COPY instruction can be use as follows:

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

Refer to the *ILE RPG for AS/400 Reference*, SC09-2508, manual for more information.

3.5.2 Replacing the *ENTRY PLIST

You can use prototypes to validate the parameters on any kind of call, not just bound calls or subprocedure calls. We replace the formal *ENTRY PLIST by a prototype/procedure interface combination. The resulting prototypes can then be used by other programs. We realize that, given the nature of our 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 56.

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

```
 * Program input parameter                                    F
D MyMainModule    PI
D  WorkDate                     D   DatFmt(*ISO)
```

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 may want to use a prototyped call to invoke our 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 MyMainModule input parameters                G
D MyMainModule    PR                ExtPgm('MAINMOD')
D  InputDate                    D   DatFmt(*ISO)
```

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 will be called by using a bound procedure call.
- If neither keyword is specified the compiler will default to EXTPROC.

For more information on these keywords, see 3.6.1.2, “External naming” on page 53. Note that is it *not* necessary that the called program uses a procedure interface for the calling program to use a prototype and a CALLP. These two 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 our system, we extract this function 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 here:

```
 * 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
P NameOfDay       B

 * procedure interface definition                                 C
D NameOfDay       PI              9
D  WorkDate                        D   Datfmt(*ISO)

  * Days of the week name table - note no field names are required
D NameData        DS
D                                 9    Inz('Monday')
D                                 9    Inz('Tuesday')
D                                 9    Inz('Wednesday')
D                                 9    Inz('Thursday')
D                                 9    Inz('Friday')
D                                 9    Inz('Saturday')
D                                 9    Inz('Sunday')
 * Define the array as an overlay of the DS name
D  Name                           9    Dim(7) Overlay(NameData)

C                   Return    Name(DayOfWeek(Workdate))

P                 E
```

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 NameofDay                             D
D NameOfDay       PR            9
D  InputDate                     D   Datfmt(*ISO)
```

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

```
 * Using subprocedure NameOfDay, Retrieve the Name of the day from
 *  WorkDate                                                           E
C                   Eval      DayName = NameOfDay(WorkDate)
```

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.

Here is what the NameOfDay subprocedure would look like 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 return a string representing the name of the day

PNameOfDay        B

DNameOfDay        PI             9
D WorkDay                        1  0 Value

 * Days of the week name table
DNameData         DS
D Data                          63    Inz('Monday   Tuesday  Wednesday-
D                                     Thursday Friday   Saturday Sunday
D Name                           9    Dim(7) Overlay(Data)

C                   Return    Name(WorkDay)

P                 E
```

Do not forget to change the prototype. More information on passing parameters to subprocedures can be found in 3.6.2, “Parameter passing styles” on page 54.

3.5.4 Using an ILE Service Program

Since these two subprocedures may 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 we need to add to these subprocedures to compile them as a separate module.

We will use the NOMAIN keyword on the H spec since this allows us 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, we stop the compiler from complaining that it cannot determine how the program will end.

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 C 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 may 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 we strongly recommend that you 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, here is what the new service program module would look like:

```
 * DATESRVPG2 from SUBPROCSRC in RPGISCOOL
 * Procedure example service program module
 *                                                         H
H Nomain

 * Include prototypes                                      D
 /Copy RPGISCOOL/SUBPROCSRC,DATESUBPR2

 * Days of the week name table
D NameData        DS
D                                9    Inz('Monday')
D                                9    Inz('Tuesday')
D                                9    Inz('Wednesday')
D                                9    Inz('Thursday')
D                                9    Inz('Friday')
D                                9    Inz('Saturday')
D                                9    Inz('Sunday')
 * Define the array as an overlay of the DS name
D  Name                          9    Dim(7) Overlay(NameData)

 * 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
P NameOfDay       B                   Export

 * procedure interface definition                          C
```

```
D NameOfDay       PI             9
D  WorkDate                       D   Datfmt(*ISO)

 * Returning Name fron NameData table Data
 *  using DayOfWeek subprocedure                                  E
C                   Return    Name(DayOfWeek(Workdate))

P                 E

 * SubProcedure:  DayOfWeek (Day of the Week)
 *  The subprocedure accept a valid date (format *ISO) and return
 *  a number (1 digit) representing the day of the week
 *  (Monday = 1, ... , Sunday = 7)                               B
P DayOfWeek       B                   Export

* procedure interface definition                                 C
D DayOfWeek       PI             1  0
D  WorkDate                       D   Datfmt(*ISO)

 * Stand Alone Fields
D AnySunday       S               D   Inz(D'1995-04-02')
D WorkNum         S              7  0
D WorkDay         S              1  0

C     WorkDate      Subdur    AnySunday     WorkNum:*D
C     WorkNum       Div       7             WorkNum
C                   Mvr                     WorkDay

 * Returning result to the calling procedure                      A
C                   If        WorkDay < 1
C                   Return    WorkDay + 7
C                   Else
C                   Return    WorkDay
C                   Endif

P                 E
```

In this example, the subprocedure DayOfWeek does not required 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 this:

```
 * DATEMAIN2 from SUBPROC in RPGISCOOL
 * Main procedure: DATEMAIN2

 * Stand Alone Fields
D DayName         S              9

 * Include prototype for Main procedure                          G
 /Copy RPGISCOOL/SUBPROCSRC,DATEMAINPR2

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

 * Program input parameter                                       F
DDateMain2        PI
D WorkDate                        D   DatFmt(*ISO)

 * Using subprocedure DayName, Retrieve the Name of the day from
 *  WorkDate                                                     E
C                   Eval      DayName = NameOfDay(WorkDate)

 * Display the content of DayName
C     DayName       Dsply

 * Terminate Program
C                Eval     *InLR = *On
```

You do not want to forget to code the prototypes. This is the code for the main procedure:

```
 * DATEMAINPR2 from SUBPROCSRC in RPGISCOOL
 * Main Program prototype                                       G
D DateMain2       Pr
D  WorkDate                      D   DatFmt(*ISO)
```

These are the ones for the subprocedures:

```
 * DATESUBPR2 from SUBPROCSRC in RPGISCOOL
 * Prototype for subprocedure DayOfWeek                         D
D DayOfWeek       PR             1  0
D  InputDate                     D   Datfmt(*ISO)

 * Prototype for subprocedure NameofDay                         D
D NameOfDay       PR             9
D  InputDate                     D   Datfmt(*ISO)
```

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/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

In this section, we show 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:

```
DOvrDBFile        PR                  ExtPgm('QCMDEXC')
D CmdString                   3000    Options(*Varsize)
D                                     Const
D CmdLength                     15P 5 Const
D CmdOpt                         3    Options(*NoPass)
D                                     Const
```

```
DCustMast         S            100
DOverride1        C                   'OVRDBF FILE('
DOverride2        C                   ') TOFILE('
DOverride3        C                   ') SHARE(*YES)'

C                   Eval      CustMast = Override1+'CustMast'+
C                                 Override2 + 'MyLibrary/CustMast' +
C                                 Override3

C                   CallP     OvrDBFile( CustMast : %Len(CustMast))

C                   Eval      *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 may 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 further information, see 3.6.2, “Parameter passing styles” on page 54.

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 will simply function 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 this keyword, refer to *ILE RPG for AS/400 Reference*, SC09-2508.

3.6.1.1 Converting the date format

In this example, we combined the CONST and DATFMT keywords. 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 is based on the example used in 3.4, “Moving from subroutines to subprocedures” on page 39, where we used the date format *ISO on all our dates definitions. Suppose that we wanted to use the DayOfWeek procedure from a program that defines its date fields as having the *USA format. All we need to do is to modify the date subprocedures so that they included the CONST keyword on the parameter definitions in the procedure interface and prototypes. Once we recompile the subprocedures and rebuild the service program, we can safely call using the *USA date field.

This is how the modified prototypes look:

```
  * DATESUBPR3 from SUBPROCSRC in RPGISCOOL
  * Prototype for subprocedure DayOfWeek
```

```
 D DayOfWeek       PR              1  0
 D  InputDate                       D   Datfmt(*ISO) Const

  * Prototype for subprocedure DayName
 D NameOfDay       PR              9
 D  InputDate                       D   Datfmt(*ISO) Const
```

We also need to modify the Procedure Interface (PI) of those two subroutines to reflect the changes made in the prototype:

```
 * DATESRVPG3 from SUBPROCSRC in RPGISCOOL
 * Procedure example service program module
 *                                                               H
H Nomain

 * Include prototypes                                            D
 /Copy RPGISCOOL/SUBPROCSRC,DATESUBPR3
```

Note: The full version of the code can be seen in 3.5.4, “Using an ILE Service Program” on page 47.

```
 * 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
P NameOfDay       B                    Export

 * procedure interface definition                                 C
D NameOfDay       PI             9
D  WorkDate                       D   Datfmt(*ISO) Const
```

Note: The full version of the code can be seen in 3.5.4, “Using an ILE Service Program” on page 47.

```
  * SubProcedure:  DayOfWeek (Day of the Week)
  *  The subprocedure accept a valid date (format *ISO) and return
  *  a number (1 digit) representing the day of the week
  *  (Monday = 1, ... , Sunday = 7)                               B
 P DayOfWeek       B                    Export

 * procedure interface definition                                 C
 D DayOfWeek       PI              1  0
 D  WorkDate                       D   Datfmt(*ISO) Const
```

Note: The full version of the code can be seen in 3.5.4, “Using an ILE Service Program” on page 47.

Here is the main procedure presented above, modified to use a *USA date format:

```
 * DATEMAIN3 from SUBPROC in RPGISCOOL
 * Main procedure: DATEMAIN3

 * Stand Alone Fields
D DayName         S             9

 * Include prototype for Main procedure                          G
 /Copy RPGISCOOL/SUBPROCSRC,DATEMAINPR3

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

 * Program input parameter                                       F
DDateMain3        PI
D WorkDate                        D   DatFmt(*USA)

 * Using subprocedure DayName, Retrieve the Name of the day from
 *  WorkDate                                                     E
C                   Eval      DayName = NameOfDay(WorkDate)

 * Display the content of DayName
C     DayName       Dsply
```

```
 * Terminate Program
C                   Eval      *InLR = *On
```

Of course, we also modified the prototype of the main procedure:

```
 DATEMAINPR3 from SUBPROCSRC in RPGISCOOL
 * Main Program prototype                                    G
D DateMain3       Pr
D  WorkDate                     D   DatFmt(*USA)
```

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')
```

3.6.1.2 External naming

Another feature of prototypes is the use of the EXTPGM and EXTPROC keywords. Note that 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. Since we 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, any calls to the program or procedure use the external name specified. If neither keyword is specified, then the external name is the prototype name, which is, the name specified in positions 7 through 21 of the Prototype PR definition and converted to uppercase.

```
DMyProcName       PR                  ExtProc('A_Really_Long_Name()')
D ProcParm1

DAProgName        PR                  ExtPGM('QODDNAME')
D Parm1

C                   CallP     MyProcName(ProcParm1)

C                   CallP     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 may 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 will export the name in mixed case as specified.

```
  * This prototype will export the name 'MixedCase'
 D MixedCase       Pr                  ExtProc('MixedCase')

  * This one will export the name 'MIXEDCASE'
 D MixedCase       Pr
```

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

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

Note

OS/400 program calls can only pass parameters by reference and by read-only reference, not by value. The keyword CONST should be use with every input parameter on the OS/400 API prototypes definition.

3.6.2.1 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 may 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.

3.6.2.2 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 may be different.
- Pass a variable that, from the caller's perspective, will not be modified.

When a parameter is passed by value, the called program or procedure can change the value of the parameter. But, the caller will never see 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))
```

3.6.2.3 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 will not be 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 would not be changed during the call.

When a parameter is passed by read-only reference, the compiler may 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 our examples of calling subprocedures have used static procedure calls, also known as bound calls. It is also possible to call subprocedures via 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 will help you to understand the basic principles involved.

3.6.3.1 Modifying the subprocedure and its prototypes

We start by modifying the prototypes to accommodate the additional subprocedure. As you can see in the following revised source, we have taken the opportunity to demonstrate how an RPG IV subprocedure can export its name in mixed-case (see also 3.6.2, “Parameter passing styles” on page 54).

```
  * DATESUBPR4 from SUBPROCSRC in RPGISCOOL

  * Prototype for subprocedure DayOfWeek
 D DayOfWeek       PR             1  0
 D  InputDate                      D   Datfmt(*ISO) Const

  * Prototype for subprocedure DayName
 D NameOfDay       PR             9    ExtProc('NameOfDay')
 D  InputDate                      D   Datfmt(*ISO) Const

  * Prototype for subprocedure NomduJour
 D NomDuJour       PR             9    ExtProc('NomDuJour')
 D  InputDate                      D   Datfmt(*ISO) Const
```

Our new subprocedure NomDuJour has an identical interface to the one for NameOfDay. This is essential since there will only be a single one invocation point. Therefore, the parameter and return value should be identical. We 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 subprocedure, 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 RPGISCOOL/SUBPRCSRC, member name DATESRVPG4.

A new prototype

One problem with using procedure pointers to call a subprocedure is that you cannot use the same prototype to describe the interface that was used in the subprocedure itself. Look at the following code, and you can see why.

Note: The markers 1 through 2 are defined immediately after this source code example.

```
  * DATESUBPR5 from SUBPROCSRC in RPGISCOOL

  * Prototype for NomduJour & NameofDay called via procedure pointer
 D Name            PR             9    ExtProc(ProcToCall@)  1
 D  InputDate                      D   Datfmt(*ISO) Const

  * Constants for procedure pointers to NameOfDay and NomDuJour
 D NomDuJour@      C                   %PAddr('NomDuJour')  2
 D NameofDay@      C                   %PAddr('NameOfDay')
```

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 subprocedure.

2 The actual procedure pointers to be used are supplied by the two constants, which specify the mixed-case names that we previously arranged to have the subprocedures 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 subprocedure based on the response. The specific changes are identified in the following code.

Note: The markers 1 through 6 are defined immediately after this source code example.

```
  * DATEMAIN4 from SUBPROCSRC in RPGISCOOL

  * Stand Alone Fields
 D DayName         S              9

  * Include prototype for Main procedure
  /Copy RPGIsCool/SubProcSrc,DATEMAINP4

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

  * Data for English/French question - default reply to French 2
 D Question        C                   'In English (E) ou Francais (F)'
 D Reply           S              1A   Inz('F')

 D ProcToCall@     S               *   ProcPtr                 3

  * Program input parameter
 D DateMain4       PI
 D  WorkDate                       D   DatFmt(*USA)

  * Ask if the response is to be in English or French          4
 C     Question      Dsply                   Reply
 C                   Select
```

```
C                   When      Reply = 'F' or Reply = 'f'       5
C                   Eval      ProcToCall@ = NomDuJour@

C                   Other
C                   Eval      ProcToCall@ = NameofDay@
C                   EndSl

 * Using the appropriate subprocedure, Retrieve the Name of the day
C                   Eval      DayName = Name(WorkDate)       6

 * Display the content of DayName
C     DayName       Dsply

 * Terminate Program
C                   Eval      *InLR = *On
```

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 will be used if the user simply presses Enter.

3 This is the definition of the procedure pointer that will be used to invoke the subprocedure. Note that 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 will be used instead.

6 The subprocedure is now invoked and the resulting name value is displayed to the user.

Try it yourself

You can recreate the above examples by using these 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:

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

3.6.3.2 Subprocedure calls

Table 1 summarizes the different types of calls and the parameter options available for each type. 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 1. Call types and options

	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
Parms by Ref.	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.

Chapter 4. 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 may 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's subprocedures as discussed in Chapter 3, "Subprocedures" on page 31.

4.1 An introduction to ILE

The Integrated Language Environment (ILE) is a new programming model as compared to the Original Programming Model (OPM). It was introduced with the operating system release V2R3 for the C language and with release V3R1 for the RPG IV language. New concepts have been introduced with the ILE programming model:

- New object types (modules, service programs, 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 1.4, "The relationship between the RPG IV language and ILE" on page 14. Modules, service programs, binding directories, activation groups, and the CL commands that pull them all together are explained in the following sections.

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 performed. A module can be composed of a main procedure (also referred to as main program) or one or more subprocedures. The term "procedure" often designates a subprocedure or a main procedure.

A module is sometimes called "compilation unit" since it comes from 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 resulting from converting 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 it may be called by ILE programs.

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.

Main procedures of modules comprising the service program are exported automatically (implicitly). The programmer does not need to use any special specifications to make a main procedure available to external users.

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 may 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:

- EXPORT(*SRCFILE) 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.

- EXPORT(*ALL) makes available all exports from the service program. Generally, we recommend 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 of 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
- New

Default activation groups exist automatically and are never deleted. There are two default activation groups. Many system programs run in the default activation group 1. RPG IV programs created with the parameter DFTACTGRP(*YES) of the CRTBNDRPG command run in the default activation group 2. In this chapter, we use the name “default activation group” for the 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. Thus, the type of an activation group is determined by the program or service program at creation time.

An activation group is created when the program is started. An activation group may 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. Data buffer and pointer to a record are part of the ODP.
- **Dynamically allocated storage**
  Temporary object created by the ALLOC operation 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 4.2.7, “Call stack and error handling” on page 99.

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 and service programs.
- **Create Service Program (CRTSRVPGM) command**
  Invokes the ILE binder (also independent of the programming language) and creates the *SRVPGM object from specified modules and other service programs.
- **Create Bound RPG Program (CRTBNDRPG) command**
  Creates a module object in the QTEMP library and creates a *PGM program object from the module. Of course, the module is lost after the job ends.

Other commands enable change, display or delete functions. Very 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 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 serve to 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 part, short examples are presented to illustrate various methods of program creation. The sample programs are intentionally simple so we can concentrate on the mechanics of binding rather than on program logic.

We use different calls: CALLB, CALLP, and function call. Differences between these types of calls are discussed in 3.3.4, “Calling your subprocedures” on page 38, and in the *ILE RPG for AS/400 Programmer's Guide*, SC09-2507.

4.2.1.1 A main procedure calls another main procedure (CALLB)

Two separate modules, M01 and M01A, are created from two source members which are bound into a program.

Module M01

Module M01 is compiled from the following source:

```
 *****************************************************************
 * M01 from ILESRC in RPGISCOOL
 *
 * CALLB to M01A bound module
 *****************************************************************

 *   Data definitions

D String          S            100A   Inz('111*1') Varying
D Position        S              5S 0
D NonDigTxt       S             30A   Inz('Non-digit in position')
D AllDigits       S             30A   Inz('All digits in string')

 *   Main procedure

C                   CallB     'M01A'
C                   Parm                    String
C                   Parm                    Position

C                   If        Position <> 0
C     NonDigTxt     Dsply                   Position
C                   Else
C     AllDigits     Dsply
C                   EndIf

C                   Eval      *InLR = *On
```

Module M01 calls module M01A with the CALLB operation. The CALLB operation uses two parameters. The second parameter Position contains the result of the call.

Module M01A

Module M01A is compiled from the following source:

```
 ****************************************************************
 * M01A from ILESRC in RPGISCOOL
 *
 * main procedure (no subprocedures)
 ****************************************************************

 *   Data definitions

D  String         S            100A   Varying
D  Position       S              5S 0

 *    Entry parameter list

C     *Entry        PList
C                   Parm                    String
C                   Parm                    Position

 *   Main procedure

C     '0123456789'  Check     String:1      Position

C                   Eval      *InLR = *On
```

Module M01A accepts two parameters from the calling module and passes the result back through the Position parameter.

The two modules, M01 and M01A, are compiled with the Create RPG Module (CRTRPGMOD) command and then bound into the program P01 by the Create Program (CRTPGM) command. This form is called *bind by copy* because the compiled code from the modules is physically copied into the resulting program. Note that no EXPORT keyword is needed for exporting the module names M01 and M01A because they are both main procedures.

Try it yourself

Compilation of the two modules can be done by using the following two commands:

```
CRTRPGMOD MODULE(RPGISCOOL/M01) SRCFILE(RPGISCOOL/ILESRC)
CRTRPGMOD MODULE(RPGISCOOL/M01A) SRCFILE(RPGISCOOL/ILESRC)
```

Then, use the following command to bind the two modules together into the program P01:

```
CRTPGM PGM(RPGISCOOL/P01) MODULE(RPGISCOOL/M01 RPGISCOOL/M01A)
ENTMOD(*FIRST) ACTGRP(QILE)
```

The ENTMOD parameter says that the *first* module, M01, gets control when the program P01 is started.

Use the following command to run the program:

```
CALL PGM(RPGISCOOL/P01)
```

4.2.1.2 A main procedure calls a subprocedure as a function

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 following source:

```
 *****************************************************************
 * M02 from ILESRC in RPGISCOOL
 *
 * function call to module M02A
 *****************************************************************

 *   Data definitions

D String          S            100A   Inz('111*1') Varying
D Position        S              3P 0
D NonDigTxt       S             30A   Inz('Non-digit in position')
D AllDigits       S             30A   Inz('All digits in string')

 *   Prototype for procedure NonDigit
 /COPY RPGISCOOL/ILESRC,CPYM02A

 *   Main procedure

C                   If        NonDigit(String : Position)
C     NonDigTxt     Dsply                   Position
C                   Else
C     AllDigits     Dsply
C                   EndIf

C                   Eval      *InLR = *On
```

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.

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 following source:

```
 *****************************************************************
 * M02A from ILESRC in RPGISCOOL
 *
 * function NonDigit
 *****************************************************************

H Nomain

 *   Procedure prototype
 /COPY RPGISCOOL/ILESRC,CPYM02A

 *   Procedure definition

P NonDigit        B                    EXPORT
 *   Procedure interface (must match the prototype)
D NonDigit        PI             1N
D  String                      100A    Value Varying
D  Position                      3P 0

C     '0123456789'  Check     String:1      Position
C                   Return    %Found

P NonDigit        E
```

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

This prototype is used as a copy member by both modules M02 and M02A:

```
 *   CPYM02A from ILESRC in RPRISCOOL
 *   Prototype for procedure NonDigit

D NonDigit        Pr              1N
D  String                       100A   Value Varying
D  Position                       3P 0
```

Try it yourself

Compilation of the two modules 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)
```

4.2.1.3 A main procedure calls a nonfunction subprocedure (CALLP)

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 following source:

```
 *****************************************************************
 * M03 from ILESRC in RPGISCOOL
 *
 * CALLP to a subprocedure in module M03A
 *****************************************************************

 *   Data definitions

D String          S            100A   Inz('111*1') Varying
D Position        S              3P 0
D NonDigTxt       S             30A   Inz('Non-digit in position')
D AllDigits       S             30A   Inz('All digits in string')

 *   Prototype for procedure NonDi

D NonDi           Pr
D                              100A   Value Varying
D                                3P 0

 *   Main procedure

C                   CallP     NonDi  (String : Position)

C                   If        Position <> 0
C     NonDigTxt     Dsply                  Position
C                   Else
```

```
C     AllDigits     Dsply
C                   EndIf

C                   Eval      *InLR = *On
```

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 the CALLP statement that does not return any 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.

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 following source:

```
 *****************************************************************
 * M03A from ILESRC in RPGISCOOL
 *
 * subprocedure - no function
 *****************************************************************

H Nomain

 *   Procedure prototype

D NonDi           Pr
D                             100A   Value Varying
D                               3P 0

 *   Procedure definition

P NonDi           B                   EXPORT
 *   Procedrure interface (must match the prototype)
D NonDi           PI
D  String                     100A   Value Varying
D  Position                     3P 0

C     '0123456789'  Check     String:1      Position

P NonDi           E
```

Note that the EXPORT keyword must be specified to make the procedure NonDi accessible to the module (main procedure) M03.

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)
```

4.2.1.4 Passing data by EXPORT and IMPORT between two modules

This method of passing data between modules 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. Such a requirement indicates poor program design.

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 following source:

```
 *****************************************************************
 * M04 from ILESRC in RPGISCOOL
 *
 * CALLB to M04A bound module - export variables
 *****************************************************************

 *   Data definitions

D String          S            100A   Inz('111*1') Varying
D                                     EXPORT
D Position        S              3P 0
D                                     EXPORT

D NonDigTxt       S             30A   Inz('Non-digit in position')
D AllDigits       S             30A   Inz('All digits in string')

 *   Main procedure

C                   CallB     'M04A'

C                   If        Position <> 0
C     NonDigTxt     Dsply                   Position
C                   Else
C     AllDigits     Dsply
C                   EndIf

C                   Eval      *InLR = *On
```

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 CALLB bound call is used to call the module M04A. Notice that EXPORT is specified in the module where the CALLB operation is used.

Module M04A

Module M04A is compiled from the following source:

```
 *****************************************************************
 * M04A from ILESRC in RPGISCOOL
 *
 * main procedure - import variables
 *****************************************************************

 *   Data definitions

D  String         S            100A   Varying
D                                     IMPORT
D  Position       S              3P 0
D                                     IMPORT

 *   Main procedure

C     '0123456789'  Check     String:1      Position

C                   Eval      *InLR = *On
```

The module M04A accepts two variables exported from the module M04 through the CALLB operation 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 may be also more difficult to debug than passing regular parameters.

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)
```

4.2.2 Creating service programs and binding by reference

The examples in this section show how the 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's 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, 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. We 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.

First, source codes of the four modules are presented starting with 4.2.2.1, "Procedure DynEdit in module M11A" on page 73. Then, different ways are shown how the modules can be combined in various service programs in the following sections:

- 4.2.2.6, "Creating one service program" on page 79
- 4.2.2.7, "Creating two chained service programs" on page 80
- 4.2.2.8, "Creating three chained service programs" on page 80

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 AS/400 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's job (if all the users are not using all the functions) may 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 may 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 4.2.3, “Binding service programs to programs” on page 81.

4.2.2.1 Procedure DynEdit in module M11A

Module M11A is compiled from the following source:

```
 ***************************************************************
 * M11A from ILESRC in RPGISCOOL
 *
 * Contains one subprocedure - DynEdit
 ***************************************************************

H Nomain
 *==============================================================
 *   DynEdit - Dynamic edit with variable decimal positions on
 *             request
 *==============================================================
 *   Procedure prototypes

 /Copy RPGISCOOL/ILESRC,CPYS11

 *   Procedure definition

P DynEdit         B                    EXPORT

 *   Procedure interface

D DynEdit         PI           100A    Varying
D  Number                       15P 0  Value
D  DecPos                        3P 0  Value

 *   Local data definitions

D  EdtNbr         S            100A    Varying
D  WorkNbr        S             30P15
D  Correction     S              1P 0
D  Pos            S              3P 0


 *   If requested decimal positions are 0 - the correction is -1
C                   If        DecPos = 0
C                   Eval      Correction = -1

 *   If requested decimal positions are > 0 - the correction is 0
C                   Else
C                   Eval      Correction = 0
C                   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....5....0
 *   111111111111111 ==> 001111111111111110000000000000

C                   Eval      WorkNbr = Number * 10 ** -DecPos

 *   Edit the work number with edit code 3 and place the result
 *   in the varying length character variable:
 *   '  1111111111111.110000000000000'

C                   Eval      EdtNbr  = %Editc(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

C                   Eval      EdtNbr = %Subst(EdtNbr : 1 :
C                                          %Len(EdtNbr) -15 +DecPos
C                                          +Correction )

 *   Return the edited number as a varying character variable

C                   Return    EdtNbr

P DynEdit         E
```

4.2.2.2 Procedure NonDigit in module M11B

Module M11B is compiled from the following source:

```
 *****************************************************************
 * M11B from ILESRC in RPGISCOOL
 *
 * Contains one procedure - NonDigit
 *****************************************************************

H 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 RPGISCOOL/ILESRC,CPYS11

 *   Procedure definition

P NonDigit        B                    EXPORT

 *   Procedure interface
D NonDigit        PI             1N
D  String                      100A   Varying
D  Position                      3P 0

 *   Check if invalid character is found in the string
 *     (other than a digit or blank)

C     ' 0123456789' Check     String:1      Position

 *   If found replace the input string with all zero digits

C                   If        %Found
C                   Eval      String = *All'0'

 *   If all digits or blanks - Replace blanks by zeros

C                   Else
C     ' ':'0'       Xlate     String        String
C                   EndIf

 *   Return *On if invalid character found, *Off if not found

C                   Return    %Found

P NonDigit        E
```

Prototype CPYS11

These prototypes are used by the copy member in modules M11, M11A, and M11B:

```
 *****************************************************************
 *
 *   CPYS11 from ILESRC in RPGISCOOL
 *
 *   Prototypes for functions: DynEdit -  Dynamic editing
 *                             NonDigit - Check for non-digit
 *                                        characters in a string
 *****************************************************************

 *   Prototype for function DynEdit - Dynamic editing

D DynEdit         Pr           100A   Varying
D  Number                       15P 0 Value
D  DecPos                        3P 0 Value

 *   Prototype for function NonDigit - Check for non-digit
 *                                     characters in a string

D NonDigit        Pr             1N
D  String                      100A   Varying
D  Position                      3P 0
```

4.2.2.3 Procedure EdtChrNbr in module M11

Module M11 is compiled from the following source:

```
 *****************************************************************
 * M11B from ILESRC in RPGISCOOL
 *
 * Contains one procedure - EdtChrNbr
 *****************************************************************
```

```
H Nomain

 *===============================================================
 *   EdtChrNbr - Edit character coded number
 *===============================================================

 *   Procedure prototypes

 /Copy RPGISCOOL/ILESRC,CPYS10
 /Copy RPGISCOOL/ILESRC,CPYS11

 *   Procedure definition

P EdtChrNbr       B                    EXPORT

 *   Procedure interface
D EdtChrNbr       PI             1N
D  CharNbr                     100A   Varying
D  DecPos                        3P 0 Value
D  EditedNbr                   100A   Varying

 *   Local data
D Pos             S              3P 0
D Number          S             15P 0


 *   Check if the characters contain all digits or blanks

C                   If        NonDigit(CharNbr : Pos)

 *   If not all digits or blanks - Return *On (error)

C                   Return    *On
C                   EndIf

 *   Else (all digits or blanks)  - Convert characters to a number

C                   Move(P)   CharNbr       Number

 *   Edit the number with requested decimal positions (edit code 3)

C                   Eval      EditedNbr = DynEdit(Number : DecPos)

 *   Return *Off (OK)

C                   Return    *Off

P EdtChrNbr       E
```

4.2.2.4 Program (main procedure) in module M10

Module M10 is compiled from the following source:

```
 ****************************************************************
 * M10 from ILESRC in RPGISCOOL
 *
 * Entering character coded numbers and edit them
 *  with requested decimal positions on the screen
 ****************************************************************
H
 *===============================================================
 *   File description - Display file
 *===============================================================

FCHRNUMW   CF   E             WorkStn

 *===============================================================
 *   Data definitions
 *===============================================================

D EditNbr         S            100A   Varying
D RC              S              1N
D CharNbr         S            100A   Varying

 *   Called procedure prototypes

 /Copy RPGISCOOL/ILESRC,CPYS10
 *   Process display file

C                   DoW       Not *InLR

 *   Show the first format to enter a character number
 *   and requested decimal positions to edit

C                   ExFmt     CHRNUMWR
C                   If        *In03
C                   Leave
C                   EndIf

 *   Edit the character coded number with requested decimal positions

C                   Eval      CharNbr = CHRNBR
C                   Eval      RC = EdtChrNbr(CharNbr: DecPos: EditNbr)

 *   If error - Supply all blanks

C                   If        RC
C                   Eval      EDTNBR = *Blanks
C                   Else

 *   Else (OK) - Trim trailing blanks and shift right

C                   EvalR     EDTNBR = %TrimR(EditNbr)
C                   EndIf

 *   Show the result on the second screen format

C                   ExFmt     CHRNUMWR2
C                   If        *In03
C                   Leave
C                   EndIf

C                   EndDo

C                   Eval      *InLR = *On
```

Prototype CPYS10

This prototype is used by the copy member in module M10 and M11:

```
 *****************************************************************
 *
 *   CPYS10 from ILESRC in RPGISCOOL
 *
 *   Prototype for function EdtChrNbr - Edit character coded number
 *
 *****************************************************************

D EdtChrNbr       Pr             1N
D  CharNbr                     100A   Varying
D  DecPos                        3P 0 Value
D  EditedNbr                   100A   Varying
```

4.2.2.5 Display file description CHRNUMW

The following display file description is used in the main procedure of module M10:

```
 ****************************************************************
 * CHRNUMW from ILESRC in RPGISCOOL
 *
 * 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 cha-
A                                      racters:'
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 numb-
A                                    er:'
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  O  6  5
A                                  8  4'Required decimal positions:'
A            DECPOS         3Y 0O  9  5EDTCDE(3)
A                                 12  4'Resulting edited number:'
A            EDTNBR        16A  O 13  5DSPATR(RI)
A                                 16  5'Press Enter.'
```

The following screens show how the 15-character coded number is edited into another character field.

Format CHRNUMWR prompts the user to enter a number as shown in Figure 2.

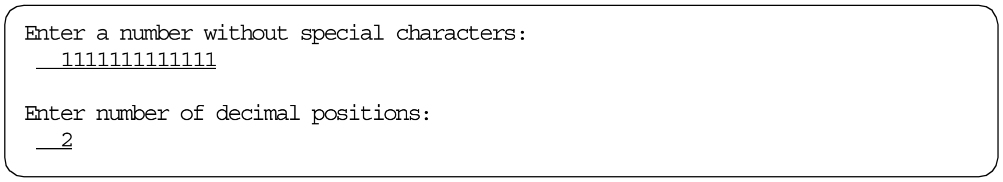

```
Enter a number without special characters:
   1111111111111

Enter number of decimal positions:
   2
```

Figure 2. Entering a character coded, nonedited number

Format CHRNUMWR2 in Figure 3 shows the result of editing along with the entered values.

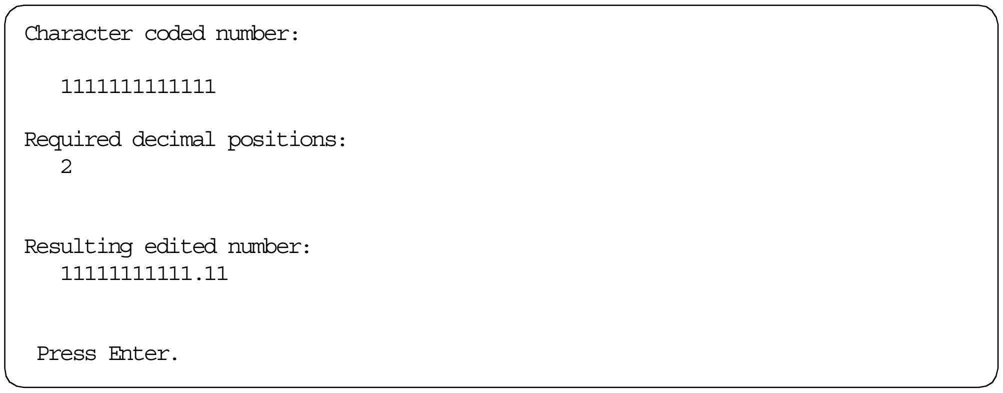

```
Character coded number:

   1111111111111

Required decimal positions:
   2


Resulting edited number:
   11111111111.11


 Press Enter.
```

Figure 3. 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 command:

```
CRTDSPF FILE(RPGISCOOL/CHRNUMW) SRCFILE(RPGISCOOL/ILESRC)

ADDLIBLE LIB(RPGISCOOL)
CRTRPGMOD MODULE(RPGISCOOL/M10) SRCFILE(RPGISCOOL/ILESRC) SRCMBR(*MODULE)
```

4.2.2.6 Creating one service program

We can bind all three procedures into one service program by using the CRTSRVPGM command:

```
CRTSRVPGM SRVPGM(RPGISCOOL/S123) MODULE(RPGISCOOL/M11A RPGISCOOL/M11B
RPGISCOOL/M11) EXPORT(*ALL) ACTGRP(*CALLER)
```

Figure 4 illustrates how the service program is built.

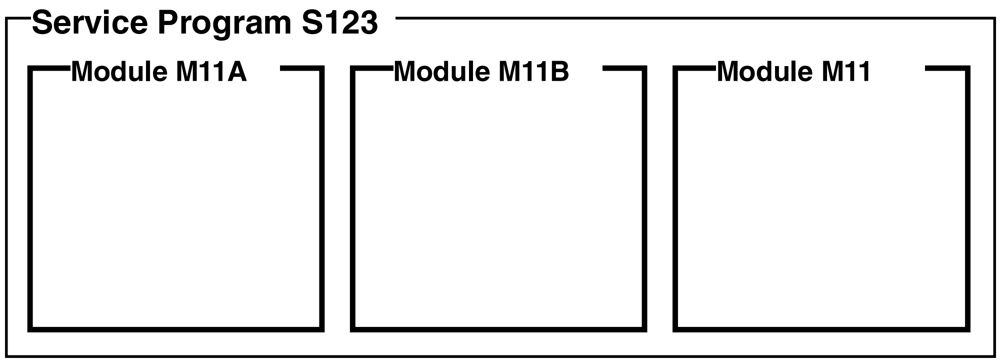


Figure 4. Service program bound from three modules

We named our service program S123 and made all its exports available to external callers. The EXPORT(*ALL) parameter specifies exactly this. Note that exports are the names of procedures DynEdit, NonDigit, and EdtCharNbr, which specify the EXPORT keyword. We shall 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 indicates that the service program will run in the same activation group as its caller. The caller is the program to which our service program is bound. Several different programs may bind the same service program and call its procedures at the same time.

4.2.2.7 Creating two chained service programs

We 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(RPGISCOOL/S12) MODULE(RPGISCOOL/M11A RPGISCOOL/M11B)
EXPORT(*ALL) ACTGRP(*CALLER)

CRTSRVPGM SRVPGM(RPGISCOOL/S3S12) MODULE(RPGISCOOL/M11) EXPORT(*ALL)
BNDSRVPGM(RPGISCOOL/S12) ACTGRP(*CALLER)
```

Figure 5 illustrates how the service programs are built.

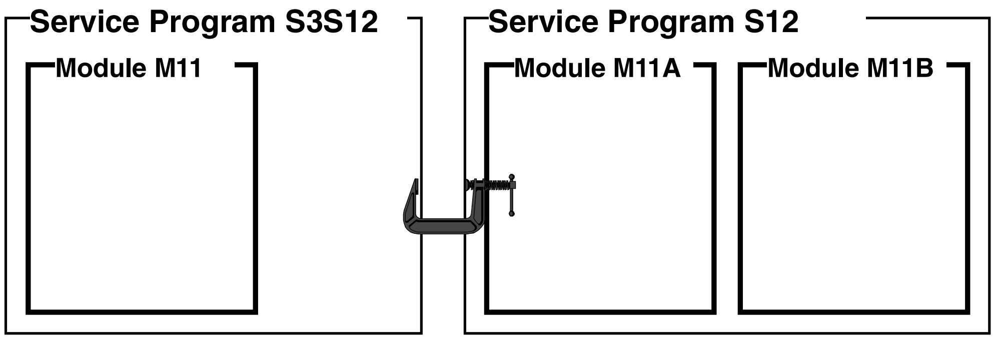


Figure 5. Two chained service programs

We named our two service programs S12 and S3S12 and let them “export all its exports” by the EXPORT(*ALL) parameter.

The *imports* from the module M11 (procedures DynEdit and NonDigit) were resolved in the example in 4.2.2.6, “Creating one service program” on page 79, 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 our previous example with service program S123.

4.2.2.8 Creating three chained service programs

Service program S3S1S2 is created if we bind module M11 with two service programs, S1 and S2. The following commands show how to do it:

```
CRTSRVPGM SRVPGM(RPGISCOOL/S1) MODULE(RPGISCOOL/M11A) EXPORT(*ALL)

CRTSRVPGM SRVPGM(RPGISCOOL/S2) MODULE(RPGISCOOL/M11B) EXPORT(*ALL)

CRTSRVPGM SRVPGM(RPGISCOOL/S3S1S2) MODULE(RPGISCOOL/M11) EXPORT(*ALL)
BNDSRVPGM(RPGISCOOL/S1 RPGISCOOL/S2) ACTGRP(*CALLER)
```

Figure 6 illustrates how the service programs are built.

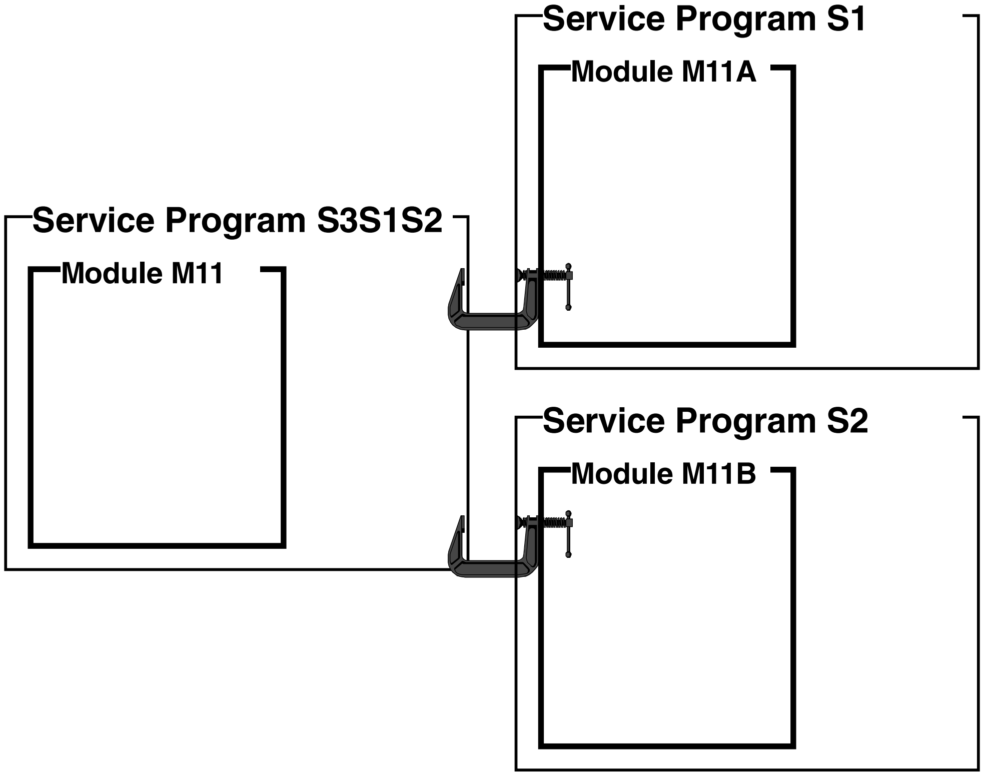


Figure 6. Three chained service programs

Note that a service program always binds at least one module and an arbitrary number of service programs.

4.2.3 Binding service programs to programs

Now that we have several arrangements of service programs, we can create several programs, each binding a different set of service programs but still each performing the same function. Only three combinations are shown.

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.

4.2.3.1 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 7 on page 82 illustrates how the program is built.

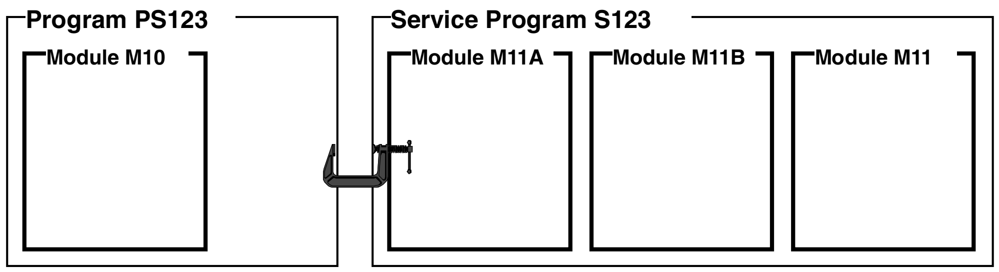


Figure 7. Program consisting of one module and one service program

Try it yourself

Use the following command to create the program object using the service program created in 4.2.2.6, "Creating one service program" on page 79:

```
CRTPGM PGM(RPGISCOOL/PS123) MODULE(RPGISCOOL/M10) ENTMOD(*FIRST)
BNDSRVPGM(S123) ACTGRP(QILE)
```

Run program PS123 using the following commands:

```
ADDLIBLE LIB(RPGISCOOL)
CALL PGM(RPGISCOOL/PS123)
```

4.2.3.2 Program from one module and a chained service program

Program PS3S12 is created by binding module M10 and the chained service program S3S12. Figure 8 illustrates how the program is built.

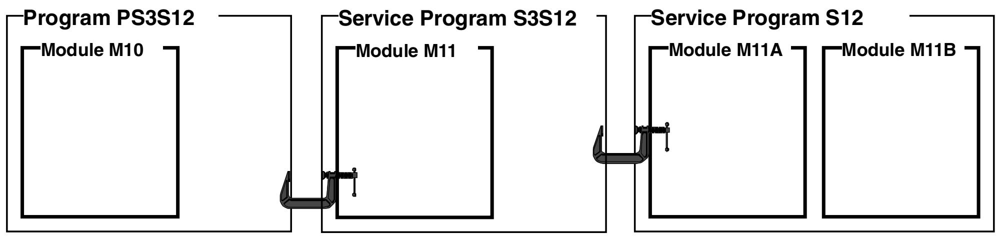


Figure 8. Program consisting of one module and two chained service programs

Try it yourself

Use the following command to create the program object using the service programs created in 4.2.2.7, "Creating two chained service programs" on page 80:

```
CRTPGM PGM(RPGISCOOL/PS3S12) MODULE(RPGISCOOL/M10) ENTMOD(*FIRST)
BNDSRVPGM(RPGISCOOL/S3S12) ACTGRP(QILE)
```

Run program PS312 by using the following commands:

```
ADDLIBLE LIB(RPGISCOOL)
CALL PGM(RPGISCOOL/PS3S12)
```

4.2.3.3 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 9 illustrates how the program is built.

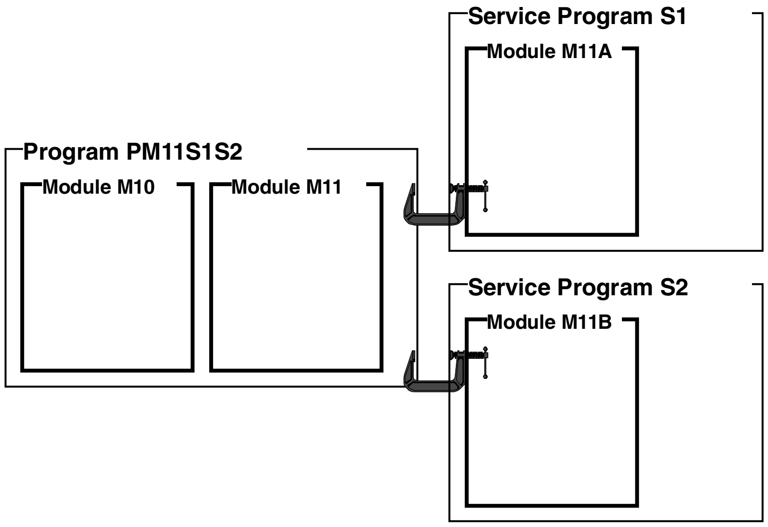


Figure 9. Program consisting of two modules and two service programs

Note that a program always binds at least one module. It does not need to bind any service program at all, or it may bind several service programs.

Try it yourself

Use the following command to create the program object using the service programs created in 4.2.2.8, “Creating three chained service programs” 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:

```
ADDLIBLE LIB(RPGISCOOL)
CALL PGM(RPGISCOOL/PM11S1S2)
```

4.2.4 Service programs, binder language, and signatures

When we created a service program, we used the EXPORT(*ALL) parameter in 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 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 UPDPGM command and specifying the service program that has changed. This way, you do not need to have all the modules in 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 AS/400 system running your application, you may 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 may 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 to the remote locations all the updated program objects in addition to the changes service program objects.

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.

4.2.4.1 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.

A short example shows two blocks of export symbols:

```
/* 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
```

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.

4.2.4.2 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 of all.
***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. We recommend that you do not use this 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. We discuss this option in 4.2.4.5, “Using your own signatures” on page 87.

4.2.4.3 Parameters of the EXPORT statement

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.

4.2.4.4 Using blocks of exported symbols

For example, our service program S12 could and should use the following binder source specified in member S12 of the QSRVSRC source file in library RPGISCOOL:

```
STRPGMEXP
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 our example S12.

The binder language source is used in the CRTSRVPGM command as follows:

```
CRTSRVPGM SRVPGM(RPGISCOOL/S12) MODULE(RPGISCOOL/M11) EXPORT(*SRCFILE)
SRCFILE(RPGISCOOL/QSRVSRC) 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 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 command:

```
RTVBNDSRC MODULE(RPGISCOOL/M11A RPGISCOOL/M11B) SRCFILE(RPGISCOOL/ILESRC)
SRCMBR(S12)
```

generates the following source text in member S12 of the ILESRC file:

```
STRPGMEXP PGMLVL(*CURRENT)
/********************************************************************/
/*   *MODULE      M11A        RPGISCOOL    06/17/99  16:28:40      */
/********************************************************************/
  EXPORT SYMBOL("DYNEDIT")
/********************************************************************/
/*   *MODULE      M11B        RPGISCOOL    06/17/99  16:28:40      */
/********************************************************************/
  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 may be specified below with 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 binder (for example, 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 may 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.

4.2.4.5 Using your own signatures

There are occasional circumstances where you may 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 will cause 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 will create 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 may want to consider explicitly specifying a signature for special circumstances. For example, you may 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's 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 may appear as wither possibility shown here:

```
STRPGMEXP PGMLVL(*CURRENT) SIGNATURE('AnIncompatibleSi')
STRPGMEXP PGMLVL(*CURRENT) SIGNATURE(X'0000000000000000F105631521336215')
```

You can determine the signature values in a service program by the command:

```
DSPSRVPGM SRVPGM(RPGISCOOL/S12) OUTPUT(*PRINT) DETAIL(*SIGNATURE)
```

The printout looks like this:

```
*...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+.
                                        Display Service Program Information
5769SS1 V4R4M0  990521
 Service program  . . . . . . . . . . . . :   S12
   Library  . . . . . . . . . . . . . . . :     RPGISCOOL
 Owner  . . . . . . . . . . . . . . . . . :   VZUPKA
 Service program attribute  . . . . . . . :   RPGLE
 Detail . . . . . . . . . . . . . . . . . :   *SIGNATURE
                                                            Signatures:
 0000000000000000F105631521336215
```

For more information about signatures, refer to *AS/400 ILE Concepts,* SC41-5606.

4.2.5 Using binding directories

Binding directories are created by two CL commands: Create Binding Directory (CRTBNDDIR) and Add Binding Directory Entry (ADDBNDDIRE) (or Work with Binding Directory Entries (WRKBNDDIRE)). The first command creates an object of *BNDDIR type. The second 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.

We create our program with the name PM11S1S2 using binding directory BM11S1S2. The binding directory BM11S1S2 is created by the following commands:

```
CRTBNDDIR BNDDIR(RPGISCOOL/BM11S1S2)
ADDBNDDIRE BNDDIR(RPGISCOOL/BM11S1S2) OBJ((RPGISCOOL/M11 *MODULE))
ADDBNDDIRE BNDDIR(RPGISCOOL/BM11S1S2) OBJ((RPGISCOOL/S1 *SRVPGM))
ADDBNDDIRE BNDDIR(RPGISCOOL/BM11S1S2) OBJ((RPGISCOOL/S2 *SRVPGM))
```

We can display contents of a binding directory by the DSPBNDDIR (or WRKBNDDIRE) command:

```
DSPBNDDIR BNDDIR(RPGISCOOL/BM11S1S2) OUTPUT(*PRINT)
```

The printout looks like this:

```
*...+....1....+....2....+....3....+....4....+....5....+....6....+....7...
                                             Display Binding Directory
5769SS1 V4R4M0  990521
 Binding Directory  . . . . . . :   BM11S1S2                       Libra
                                               --------Creation---------
 Object         Type           Library        Date           Time
 M11            *MODULE        *LIBL          06/08/99       10:10:43
 S1             *SRVPGM        *LIBL          06/11/99       09:07:42
 S2             *SRVPGM        *LIBL          06/11/99       09:07:49
```

Now, we can create our program by the binder command:

```
CRTPGM PGM(RPGISCOOL/PM11S1S2) MODULE(RPGISCOOL/M10)
BNDDIR(RPGISCOOL/BM11S1S2) ACTGRP(QILE)
```

This creates the same binding as the earlier example in 4.2.3.3, “Program from two modules and two service programs” on page 83. The program still performs the same function as our 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 may display your module contents by the command:

```
DSPMOD MODULE(RPGISCOOL/M11B) DETAIL(*IMPORT)
```

Then, you would get the following system procedure names:

```
_QRNX_SUBP_EXCP
_QRNX_INIT_H
_QRNX_INIT
_QRNX_SIGNAL_EXCP
Q LE leDefaultEh
```

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.

We 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.

4.2.6 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).

4.2.6.1 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. This is a default value for CRTPGM command. Accepting this default for all ILE programs will 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.
name	A named activation group is created if the program is called and the activation group does not yet exist. This is a default value for the CRTBNDRPG command, using the name QILE. This is the most preferable option if there’s 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 as been invoked in a NAMED activation group.

The Create Service Program (CRTSRVPGM) command allows the same parameter values except *NEW:

name	A named activation group is created if the service program’s procedure is called and the activation group does not already exist.
***CALLER**	No new activation group is created. The service program’s 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.

4.2.6.2 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) that 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 an RCLACTGRP command to end the activation group as you would have to with a named activation group.
  - A *named* activation group exists during all 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 Reclaim Activation Group (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 may 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.
  - *Default* activation group is never ended before the job ends.

4.2.6.3 Overriding files in activation groups

If you want to change some characteristics of a file (database, printer, display, etc.), you can use an OVRxxxF command, where xxx stands in place of DB for database or PRT for printer, etc. For example, you can use the Override with 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 (boundaries) 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:

***ACTGRPDFN**
: 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:

***ACTGRPDFN**
: 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:

***ACTGRPDFN**
: 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.

4.2.6.4 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 command. Temporary specification is done by using the OVRxxxF (override) command.

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 will no longer be positioned at that record. It will be 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 will return record 7. However, if a second program sharing the ODP is called and it chains to record 11, the first program will read 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 will no longer retain 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

4.2.6.5 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 10 shows.

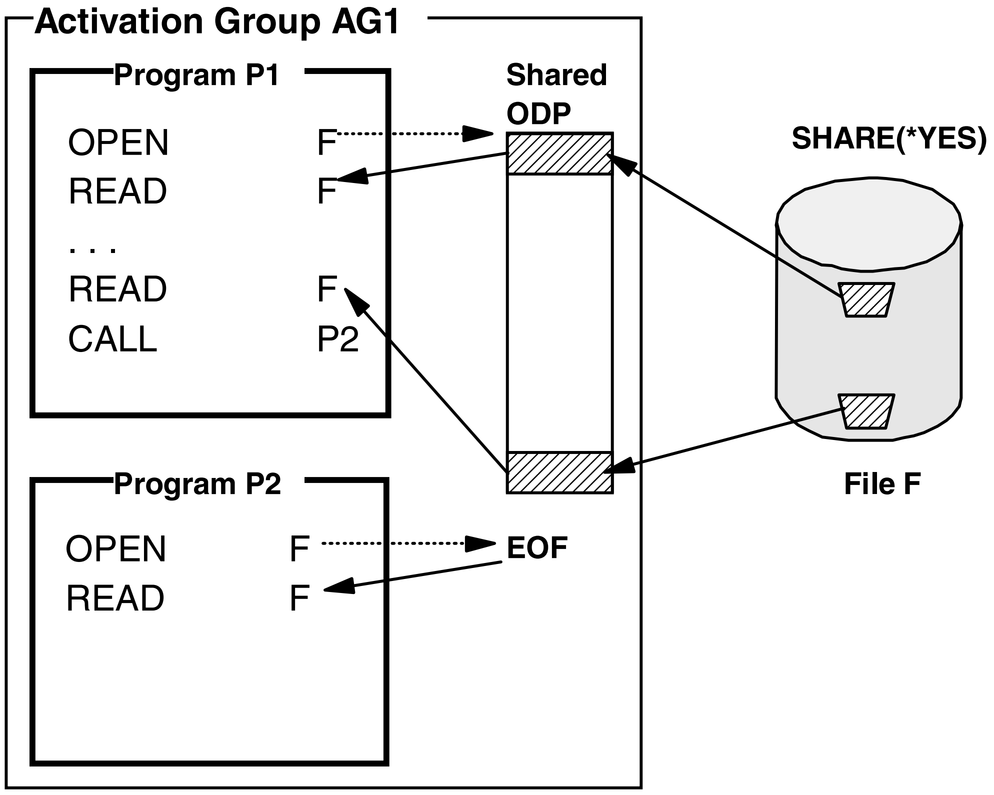


Figure 10. Shared open data path in a common activation group

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 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 would look better. If the program P2 comes from a different vendor than program P1, it could 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 at opening the file in program P2.

Program P1

Program P1 is compiled from the following source:

```
*****************************************************************
* P1 from ILESRC in RPGISCOOL
*
* Sequential processing of a file
*****************************************************************
```

```
H DFTACTGRP(*NO) ACTGRP('AG1')  1

 *   File descriptions

FITEMS     UF   E             Disk
FREPORT    O    E             Printer

 *   Main procedure

C                   Read(N)   ITEMS

C                   If        %EoF  2
C                   Eval      EOFTEXT = 'P1 End of file'
C                   Else
C                   Eval      EOFTEXT = 'P1 Beginning of file'
C                   EndIf
C                   Write     EOFLINE

C                   DoW       Not %EoF  3
C                   Write     ITEMDETAIL
C                   Read(N)   ITEMS
C                   EndDo

C                   Call      'P2'    4

C                   Eval      *InLR = *On
C                   Return
```

Note the following points:

1 Program P1 runs in activation group AG1. It is created by the CRTBNDRPG command.

2 After the first READ operation, the program tests if 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, P2.

Program P2

Program P2 is compiled from the following source:

```
 *****************************************************************
 * P2 from ILESRC in RPGISCOOL
 *
 * Sequential processing of a file
 *****************************************************************

H DFTACTGRP(*NO) ACTGRP('AG1')   1

 *   File descriptions

FITEMS     UF   E             Disk
FREPORT    O    E             Printer

 *   Main procedure

C                   Read      ITEMS

C                   If        %EoF 2
C                   Eval      EOFTEXT = 'P2 End of file'
C                   Else
C                   Eval      EOFTEXT = 'P2 Beginning of file'
C                   EndIf
C                   Write     EOFLINE

C                   DoW       Not %EoF  3

C                   Eval      UNITPR = UNITPR * 1,01
C                   Update    ITEMSR

C                   Write     ITEMDETAIL
```

```
C                   Read      ITEMS
C                   EndDo

C                   Eval      *InLR = *On
C                   Return
```

Note the following explanations:

1 Program P2 runs in the same activation group, AG1, as program P1.

2 Now, the current record position is still at EOF because the database file is shared. Note that it is true, even if program P1 ended with LR indicator *ON. The program prints an `End of file` message.

3 The reading cycle is therefore skipped, and program P2 ends and returns to the program P1 which also ends.

You can recognize that the programs must be compiled by the CRTBNDRPG command (option 14 in PDM) because they specify keywords DFTACTGRP and ACTGRP in the H spec. These keywords are not allowed in the CRTRPGMOD command (option 15 in PDM) because the ACTGRP parameter must be specified only when creating a *PGM object. If using a CRTRPGMOD command, the activation group would be specified on the subsequent CRTPGM command.

File ITEMS

This example uses a physical file member with the following definition:

```
 *****************************************************************
 * ITEMS from ILESRC in RPGISCOOL
 *
 * Item master file
 *****************************************************************
A                                      UNIQUE
A          R ITEMSR
 *   Item number
A            ITEMNBR        5          COLHDG('Item' 'number')
A            UNITPR         9  2       COLHDG('Unit' 'price')
 *   Item description
A            ITEMDESC      50          COLHDG('Item description')

 *   Key field
A          K ITEMNBR
```

Here is the content of the sample physical file used in our examples:

```
Item           Unit    Item description
number         price
00001          26.78   First item
00002          53.59   Second item
00003          80.38   Third item
```

Printer file REPORT

This printer file definition is used in our sample programs:

```
 *****************************************************************
 *   REPORT from ILESRC in RPGISCOOL
 *
 *   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

Create the programs P1 and P2 by using the following commands. Prior to creating the programs, the physical file and printer file must be created:

```
ADDLIBLE 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, use the following command:

```
CALL PGM(RPGISCOOL/P1)
```

After running the example by calling program P1 from the command line, we 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
```

We 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 could 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(RPGISCOOL/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 will be shared between programs P1 and P2, and the results will be the same as before.

If we change the source type to CLLE (ILE CL), we can compile the same CL program with the Create Bound CL Program (CRTBNDCL) command. We 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.

4.2.6.6 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 illustrated in Figure 11. Note that if another program using the ITEMS file were to be called in activation group AG1, it could share the ODP created by program P1A.

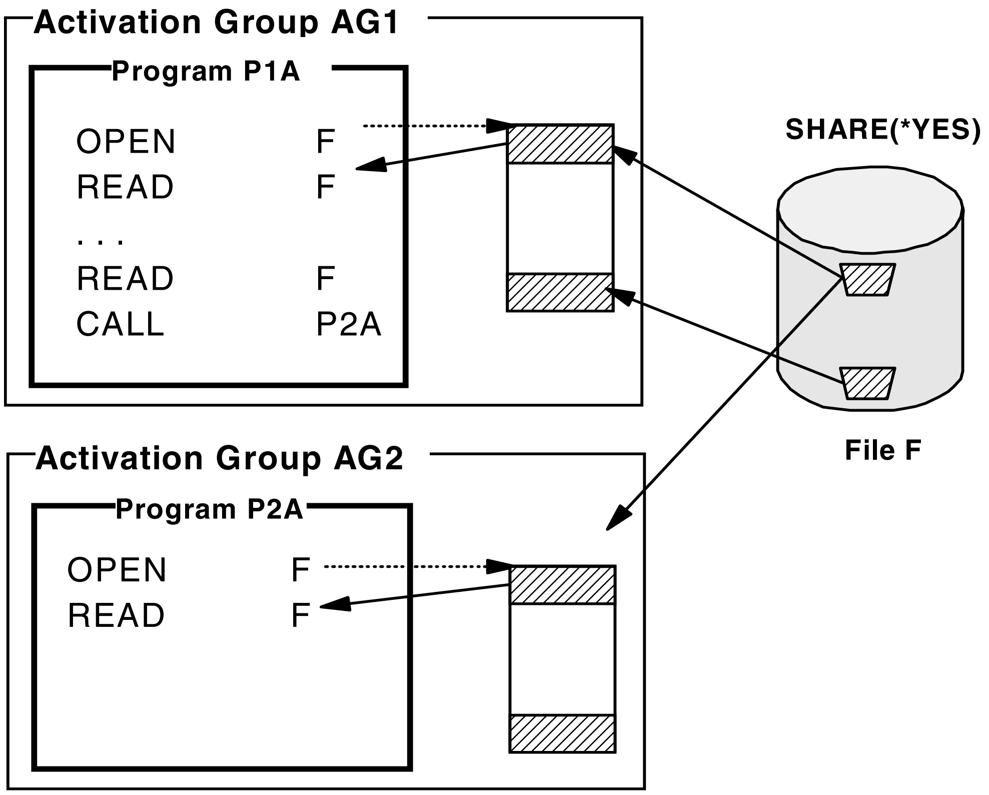


Figure 11. Open data paths in different activation groups

Note these points:

- Program P1A runs in activation group AG1 and ends its processing at EOF. This was the same for program P1 from the preceding example.
- 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 more 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:

```
 *****************************************************************
 * P1A from ILESRC in RPGISCOOL
 *
 * Sequential processing of a file
 *****************************************************************

H DFTACTGRP(*NO) ACTGRP('AG1')

 *   File descriptions

FITEMS     IF   E             Disk
FREPORT    O    E             Printer

 *   Main procedure
```

```
C                   Read      ITEMS

C                   If        %EoF
C                   Eval      EOFTEXT = 'P1A End of file'
C                   Else
C                   Eval      EOFTEXT = 'P1A Beginning of file'
C                   EndIf
C                   Write     EOFLINE

C                   DoW       Not %EoF
C                   Write     ITEMDETAIL
C                   Read      ITEMS
C                   EndDo

C                   Call      'P2A'

C                   Eval      *InLR = *On
C                   Return
```

Program P2A

Program P2A is identical to P2, except it specifies activation group AG2 in its H specification and prints slightly different text:

```
 ****************************************************************
 * P2A from ILESRC in RPGISCOOL
 *
 * Sequential processing of a file
 ****************************************************************

H DFTACTGRP(*NO) ACTGRP('AG2')

 *   File descriptions

FITEMS     UF   E             Disk
FREPORT    O    E             Printer

 *   Main procedure

C                   Read      ITEMS

C                   If        %EoF
C                   Eval      EOFTEXT = 'P2A End of file'
C                   Else
C                   Eval      EOFTEXT = 'P2A Beginning of file'
C                   EndIf
C                   Write     EOFLINE

C                   DoW       Not %EoF

C                   Eval      UNITPR = UNITPR * 1,01
C                   Update    ITEMSR

C                   Write     ITEMDETAIL

C                   Read      ITEMS
C                   EndDo

C                   Eval      *InLR = *On
C                   Return
```

After running the example by calling program P1A from the command line, we 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
```

We 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 opened its own data path in a newly created activation group AG2 no matter if program P1A already has its 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(*CALLVL) does not cause the ODP to be shared.

If you really want to share the ODP in both activation groups, you would have to issue the following commands, the open scope being the entire job:

```
OVRDBF     FILE(ITEMS) OVRSCOPE(*JOB) SHARE(*YES) OPNSCOPE(*JOB)
CALL       PGM(P1A)
```

The OVRSCOPE parameter says that sharing the ITEMS file will last for the entire time the job is active (unless another override comes in between). The OPNSCOPE says that subsequent opens of the ITEMS file after the first open will follow the attributes set by this command. The attributes are, among others, open options for all types of access (input, output, update, delete) and, of course, the share option set by the override command just before.

Try it yourself

You can create the programs P1A and P2A using the following commands. Prior to creating the programs, the physical file and printer file must be created (if they weren't 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, use the following command:

```
CALL PGM(RPGISCOOL/P1A)
```

4.2.7 Call stack and error handling

The program stack known from OPM has been renamed to "call stack" because not only programs but also subprocedures are contained in it.

4.2.7.1 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 program entry procedure is a piece of code that is generated by the compiler to initialize (activate) the program. The program entry procedure passes control to the program code that was written by the programmer.

Service programs, as well as OPM programs, have no PEP. Subprocedures and modules that are called by the bound call have no PEP either.

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 last part of the call stack is shown in Figure 12.

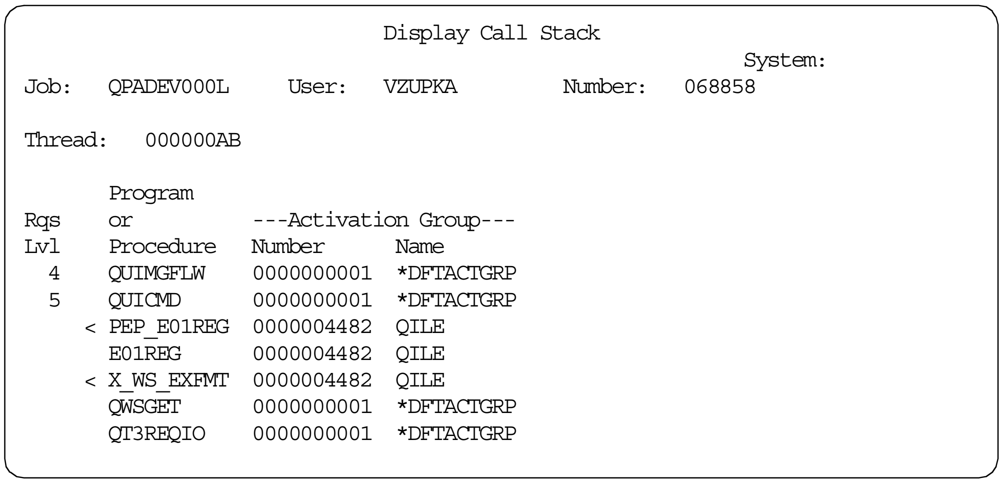

```
                              Display Call Stack
                                                        System:
Job:   QPADEV000L     User:   VZUPKA         Number:   068858

Thread:   000000AB

       Program
Rqs    or          ---Activation Group---
Lvl    Procedure   Number      Name
  4    QUIMGFLW    0000000001  *DFTACTGRP
  5    QUICMD      0000000001  *DFTACTGRP
     < PEP_E01REG  0000004482  QILE
       E01REG      0000004482  QILE
     < X_WS_EXFMT  0000004482  QILE
       QWSGET      0000000001  *DFTACTGRP
       QT3REQIO    0000000001  *DFTACTGRP
```

Figure 12. Call stack for program E01REG

Program E01REG is waiting in the EXFMT operation. The entry QWSGET is a control boundary. The entry < PEP_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 PEP_E01REG, and press F22. The full name is _QRNP_PEP_E01REG.

4.2.7.2 Error handling in ILE

If an error message arises in a procedure, it is processed by various program functions in the following order:

1. Error resulting indicator (in the "Lo" column) or the (E) modifier in some operations that can be tested by the %ERROR built-in function handles the error message.
2. ILE condition handler can handle escape, status, and notify error messages coming from any procedure.

3. INFSR subroutine (file information subroutine) handles the error messages resulting from file I/O operations. The INFSR keyword cannot be specified if the file is to be accessed by a subprocedure, or if NOMAIN is specified on the H specification.
4. *PSSR subroutine (program status subroutine). A *PSSR can be defined in a subprocedure, and each subprocedure can have its own *PSSR. Note that 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. Default RPG exception handler.

The ILE condition handler is a new function with ILE, while 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 (and without the "Lo" resulting indicator), 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 could catch error messages 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, 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.

For *STATUS and *NOTIFY error message types, see *ILE Concepts,* SC41-5606.

4.2.7.3 ILE condition handler interface

ILE conditions are OS/400 exception messages represented in a manner independent of the system. An ILE condition token 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 2) that tells the kind of message it received and from which procedure.

Table 2. Input parameters for the condition handler

Variable name	Description	RPG IV data type	Values
CondToken	Incoming message identification	Char(12)	See Table 3
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 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 13 describes this structure. The same structure (as seen in Table 3) applies to the NewToken.

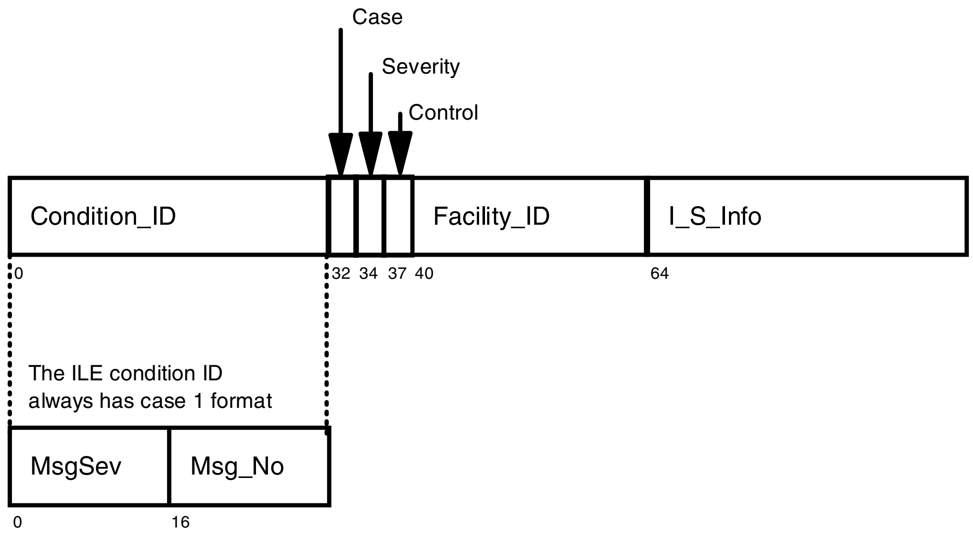


Figure 13. ILE condition token layout

Table 3. Condition token structure

Variable name	Description	RPG IV data type	Values
MsgSev	Message severity	Integer(5)	0, 1, 2, 3, 4
MsgNo	Message number	Integer(5) or Char(2)	For example, X'1211' for divide by zero
CaseSevCtl	Case, Severity, Control in one byte	Char(1)	See note
MsgType or FacilityID	Message type	Char(3)	For example, CEE, CPF, MCH RNX
MsgKey or ISInfo	Unique message key	Char(4)	A unique binary code

Note: Case, Severity, and Control are binary numbers 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. Actually used numbers are 0 to 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.

Message type and message number uniquely identify the incoming message.

You can process a message so that you replace it by 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 program builds a new message according to the parameters listed in Table 4.

Table 4. Parameters for the CEENCOD 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 Q*USR*MSG must be created using the Create Message File CRTMSGF command to enable promotion of USRxxxx messages. Message descriptions prefixed with *USR* must be added in the message file using the ADDMSGD command. You can change the message file name to Q*xxx*MSG, but the messages must begin with *xxx*. This is because you cannot specify the message file name in the parameters.

Having this information, the CEENCOD 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.

4.2.7.4 An example of the condition handler

A condition handling program is illustrated under the name ILEERRHDL in the following example. The data definition part of the program is shown first.

```
 ****************************************************************
 *
 *   Member ILEERRHDL from ILESRC in RPGISCOOL
 *
 *   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.
 *
 ****************************************************************
 H

 *==============================================================
 *   Data definitions
 *==============================================================

 * Program prototypes                                      1
 /COPY RPGISCOOL/ILESRC,ILEERRPR

 *   Input and output parameters                     5
D IleErrHdl       PI
```

```
D  CondToken                    12A
D  ProcName                     10A
D  RtnAct                       10I 0
D  NewToken                     12A

 *   Message information structure (CondToken and NewToken)        2

D CondTokenPtr    S               *

D CondTokenDS     DS                   based(CondTokenPtr)
D  MsgSev                        5I 0
D  MsgNum                        2A
D  CaseSevCtl                    1A
D  MsgType                       3A
D  MsgKey                        4A

 *   Constants                                                        3

D ResumeNextMI    C                   Const(10)
D PercolCallStk   C                   Const(20)
D PercolNextHnd   C                   Const(21)
D PromoteCallStk  C                   Const(30)
D PromoteNextHnd  C                   Const(31)

 *   Parameter data for CEENCOD procedure          4

D MsgSeverity     S              5I 0 Inz(4)
D Case            S              5I 0 Inz(1)
D Severity        S              5I 0 Inz(4)
D Control         S              5I 0 Inz(0)
D FacilityID      S              3A   Inz('USR')
```

Note the following explanation:

1 Parameters for the CEENCOD ILE 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 program are initialized. MsgNumber an FacilityId variables are overwritten dynamically.

5 Entry parameters for the condition handler are specified as an *ENTRY parameter list because the condition handler is not a prototyped procedure.

The procedural part of the ILEERRHDL program tests for error messages and takes appropriate actions.

```
 *=================================================================
 *   Mainline program
 *=================================================================

C                   Eval      CondTokenPtr = %addr(CondToken)

C                   Select

 *   Error RNX1218 Unable to allocate a record in file xxx...      1

C                   When      MsgType = 'RNX' and MsgNum = X'1218'
C                   CallP     ILECondHandler(MsgSeverity : MsgNum : Case :
C                              Severity : Control : FacilityID :
C                              MsgKey : NewToken : *Omit)
C                   Eval      RtnAct = PromoteCallStk

 *   Error RNX1021 Attempt to write a duplicate record to file xxx...   2

C                   When      MsgType = 'RNX' and MsgNum = X'1021'
C                   CallP     ILECondHandler(MsgSeverity : MsgNum : Case :
C                              Severity : Control : FacilityID :
C                              MsgKey : NewToken : *Omit)
C                   Eval      RtnAct = PromoteNextHnd
```

```
      *   Error MCH1211                                               3
      *   Attempt made to divide by zero for fixed point operation.
     C                   When      MsgType = 'MCH' and MsgNum = X'1211'
     C                   Eval      RtnAct = ResumeNextMI
      *                                                                          4
     C                   Other
     C                   Eval      RtnAct = PercolNextHnd

     C                   EndSl

     C                   Return
```

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

This file contains the ILE condition handler API prototype used in this example. It must exist prior to creating any of the modules used in this example. A good description of the ILE condition handler APIs can be found in *System API Reference - OS/400 Integrated Language Environment (ILE) CEE APIs*, SC41-5861.

```
      * ILEERRPR from ILESRC in RPGISCOOL
      *---------------------------------------------------------------
      *   Input parameters
     D IleErrHdl       PR                  ExtPgm('ILEERRHDL')
     D  CondToken                    12A
     D  ProcName                     10A
     D  RtnAct                       10I 0
     D  NewToken                     12A

      *   Parameters for CEENCOD procedure (own message construction)
     D ILECondHandler  PR                  ExtProc('CEENCOD')
     D  MsgSeverity                   5I 0
     D  MsgNumber                     2A
     D  Case                          5I 0
     D  Severity                      5I 0
     D  Control                       5I 0
     D  FacilityID                    3A
     D  MsgKey                        4A
     D  NewToken                     12A
     D  Feedback                     12A   Options(*Omit)

     D RegCondHdlr     PR                  ExtProc('CEEHDLR')
     D  CondHdlr@                      *   PROCPTR
     D  ProcName                     10A
     D  Feedback                     12A   Options(*Omit)

     D UnRegCondHdlr   PR                  ExtProc('CEEHDLU')
     D  CondHdlr@                      *   PROCPTR
     D  Feedback                     12A   Options(*Omit)
```

Try it yourself

Create this module by using the following command:

```
CRTRPGMOD MODULE(RPGISCOOL/ILEERRHDL) SRCFILE(RPGISCOOL/ILESRC)
SRCMBR(*MODULE)
```

This module is used in 4.2.7.6, "Creating a program that uses the condition handler" on page 109.

4.2.7.5 A program that uses the condition handler

The following 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 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 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.

An exercise for you

Notice the variable CondHdlr@ used in the following 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.

First, the data definition part of the E01REG module is shown:

```
 *****************************************************************
  *  E01REG from ILESRC in RPGISCOOL
  *
  *  Errors, condition handler registration
 *****************************************************************

FSTOCK     UF A E           K Disk
FSTOCKW    CF   E             WorkStn

 * Program prototypes                 3
 /COPY RPGISCOOL/ILESRC,ILEERRPR
 *================================================================
 *   Data definitions
 *================================================================

 *   Procedure pointer to ILEERRHDL program
D CondHdlr@       S               *   Procptr 1
D                                     Inz(%Paddr('ILEERRHDL'))


 *   PSDS data structure (this procedure name)
D                SDS                  NOOPT
D ProcName          *PROC                   2
```

```
 *   Key list for STOCK file

C     Key           KList
C                   KFld                    STOCKNO
C                   KFld                    ITEMNBR
```

Note the following points:

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 will be 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 106.

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 106.

The relevant parts of the procedural part of the E01REG module are shown here:

```
 *=================================================================
 *   Mainline program
 *=================================================================

 *   Register an ILE condition handler program                 1
C                   CallP     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.
 *
C                   CallB     'E01ITEMS'   2
C                   Parm                    ITEMNBR
C     *In80         Parm                    RC                1
...

 *   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.

C     Key           Chain     STOCK 3
...

 *   Unregister the ILE condition handler program              4
C                   CallP     UnregCondHdlr(CondHdlr@ : *Omit)
C                   Eval      *InLR = *On
C                   Return
```

Note these points:

1 The E01REG module registers the condition handler by the CALLB bound call with the parameter list.

2 After some processing, a check for item number is made by calling the E01ITEMS module. In the module, a divide by zero error occurs (see the following source for E01ITEMS). The condition handler practically ignores this message.

3 After some more processing, a record is read from the STOCK file for update. If another job holds the same record locked, 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, the E01ITEMS module is shown here:

```
 *****************************************************************
 *  E01ITEMS from ILESRC in RPGISCOOL
 *
 *  E01ITEMS - Check if an item is in ITEMS file
 *****************************************************************

FITEMS     IF   E           K Disk

D RC              S              1N
D Divisor         S             15P 0 Inz(*Zero)

 *   Entry parameter list
C     *Entry        PList
C                   Parm                    ITEMNBR
C                   Parm                    RC

 *   Look up the item number
C     ITEMNBR       SetLL     ITEMS
 *   If not found - Set RC *ON else set RC *OFF
C                   Eval      RC = Not %Found

 *   Generate an artificial error (divide by zero)
C                   Eval      Divisor = 0
C                   Eval      UNITPR = UNITPR / Divisor

C                   Return
```

Try it yourself

You can create these two modules by using the two following commands. The two physical files and the display file used in this example must have been created prior to creating the modules, as described in 4.2.7.8, “Files used in the condition handling example” on page 113.

```
CRTRPGMOD MODULE(RPGISCOOL/E01REG) SRCFILE(RPGISCOOL/ILESRC) SRCMBR(*MODULE)
CRTRPGMOD MODULE(RPGISCOOL/E01ITEMS) SRCFILE(RPGISCOOL/ILESRC)
SRCMBR(*MODULE)
```

These modules are used in the following section.

4.2.7.6 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, therefore, not likely to be useful in other programs. It will most likely be bound by copy into the program with the application modules.

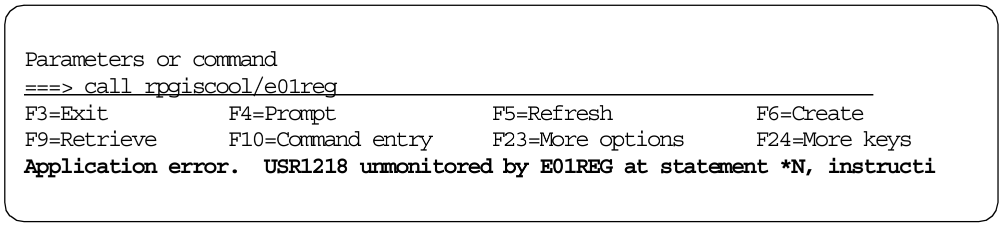
However, in our example, we tried to make our condition handler fairly generic so that this same handler would be useful for many different programs. Therefore, it makes more sense to place the condition handler in a service program. In a real application environment, you would most likely place it in a service program together with other modules, perhaps even other ILE condition handling modules. However, for purposes of simplicity in our example, we place our condition handler module 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)
```

4.2.7.7 Running a program that uses the condition handler

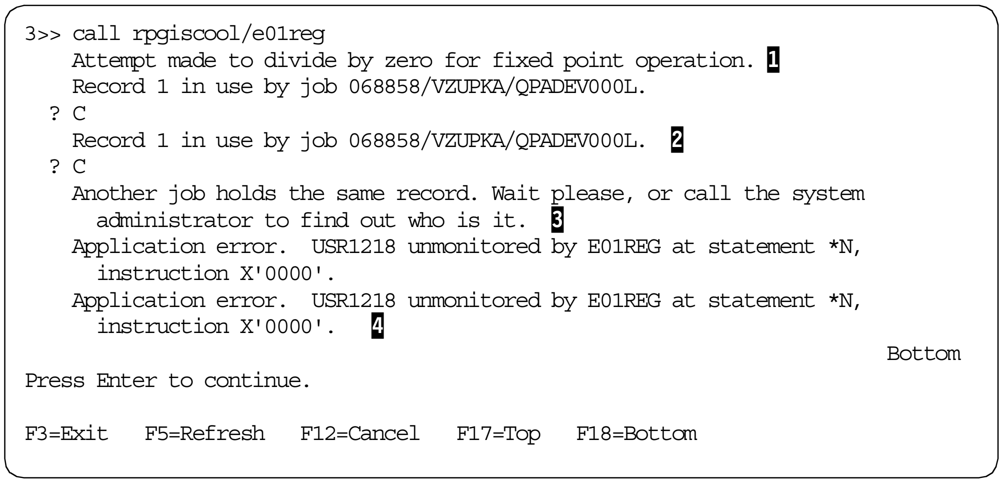
We call the program E01REG from the command line in a session. We let a record be locked while being displayed on the screen. Then, we call the same program from another session and try to display the same record. After some time (60 seconds is default), the second call ends with a message on the command line as Figure 14 shows.



```
Parameters or command
===> call rpgiscool/e01reg
F3=Exit          F4=Prompt             F5=Refresh            F6=Create
F9=Retrieve      F10=Command entry     F23=More options      F24=More keys
Application error.  USR1218 unmonitored by E01REG at statement *N, instructi
```

Figure 14. An error message after running a program with a condition handle

If we display the job log, the screen shown in Figure 15 appears.



```
3>> call rpgiscool/e01reg
    Attempt made to divide by zero for fixed point operation. 1
    Record 1 in use by job 068858/VZUPKA/QPADEV000L.
  ? C
    Record 1 in use by job 068858/VZUPKA/QPADEV000L.  2
  ? C
    Another job 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,
      instruction X'0000'.
    Application error.  USR1218 unmonitored by E01REG at statement *N,
      instruction X'0000'.  4
                                                                     Bottom
Press Enter to continue.

F3=Exit   F5=Refresh   F12=Cancel   F17=Top   F18=Bottom
```

Figure 15. Job log messages resulting from a program call with errors

Let us look closer at some of the messages:

1 The divide by zero was generated by the module E01ITEMS in the service program SE01ITEMS as Figure 16 shows.

2 Message `Record 1 in use...` is sent by the system to module E01REG as seen in Figure 17.

3 The message `Another job holds the same record...` is our message USR1218 that replaced the message RNX1218 (which doesn't appear in the job log). Details are shown in Figure 18 on page 112.

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 19 on page 112 shows the message details. This is the message displayed on the message line.

```
Message ID . . . . . . :   MCH1211       Severity . . . . . . . :   40
Date sent  . . . . . . :   06/16/99      Time sent  . . . . . . :   16:04:00
Message type . . . . . :   Escape
CCSID  . . . . . . . . :   65535

From program . . . . . . . . . :   SE01ITEMS
  From library . . . . . . . . :     RPGISCOOL
  From module  . . . . . . . . :     E01ITEMS
  From procedure . . . . . . . :     E01ITEMS
  From statement . . . . . . . :     23

To program . . . . . . . . . . :   SE01ITEMS
  To library . . . . . . . . . :     RPGISCOOL
  To module  . . . . . . . . . :     E01ITEMS
  To procedure . . . . . . . . :     E01ITEMS
  To statement . . . . . . . . :     23
```

Figure 16. Details of the divide by zero message

```
Message ID . . . . . . :   CPF5027       Severity . . . . . . . :   30
Date sent  . . . . . . :   06/16/99      Time sent  . . . . . . :   16:04:05
Message type . . . . . :   Notify
CCSID  . . . . . . . . :   65535

From program . . . . . . . . . :   QDBSIGEX
  From library . . . . . . . . :     QSYS
  Instruction  . . . . . . . . :     014A

To program . . . . . . . . . . :   E01REG
  To library . . . . . . . . . :     RPGISCOOL
  To module  . . . . . . . . . :     E01REG
  To procedure . . . . . . . . :     E01REG
  To statement . . . . . . . . :     102
```

Figure 17. Details of the record in use message

```
Message ID . . . . . . :   USR1218       Severity . . . . . . . :   00
Date sent  . . . . . . :   06/16/99      Time sent  . . . . . . :   16:04:06
Message type . . . . . :   Escape


From program . . . . . . . . . :   E01REG
  From library . . . . . . . . :     RPGISCOOL
  From module  . . . . . . . . :     E01REG
  From procedure . . . . . . . :     E01REG
  From statement . . . . . . . :     102

To program . . . . . . . . . . :   E01REG
  To library . . . . . . . . . :     RPGISCOOL
  To module  . . . . . . . . . :     E01REG
  To procedure . . . . . . . . :     _QRNP_PEP_E01REG
  To statement . . . . . . . . :     *N
```

Figure 18. Details of the USR1218 message

```
Message ID . . . . . . :   CEE9901       Severity . . . . . . . :   30
Date sent  . . . . . . :   06/16/99      Time sent  . . . . . . :   16:04:06
Message type . . . . . :   Escape
CCSID  . . . . . . . . :   65535

From program . . . . . . . . . :   QLEAWI
  From library . . . . . . . . :     QSYS
  From module  . . . . . . . . :     QLEDEH
  From procedure . . . . . . . :     Q LE leDefaultEh
  From statement . . . . . . . :     232

To program . . . . . . . . . . :   QUOCMD
  To library . . . . . . . . . :     QSYS
  Instruction  . . . . . . . . :     01DA
```

Figure 19. Details of the CEE9901 message

Since we did not leave any error unhandled by the condition handler, no CPF9999 Function Check message appeared.

If we specified result code 10 (ResumeNextMI) for "other" messages in the condition handler, our job log would appear as shown in Figure 20, for example:

```
C                   Eval      RtnAct = ResumeNextMI
```

Now, the function check escape messages are processed by the condition handler according to the rules in 4.2.7.2, “Error handling in ILE” on page 100.

```
3 > call rpgiscool/e01reg
    Attempt made to divide by zero for fixed point operation.
    Record 1 in use by job 068858/VZUPKA/QPADEV000L.
  ? C
    Record 1 in use by job 068858/VZUPKA/QPADEV000L.
? C
    Another program holds the same record. Wait please, or call the system
      administrator to find out who is it.
    Function check. USR1218 unmonitored by E01REG at statement *N, instruction
      X'0000'.
    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 20. Job log with function check messages

Try it yourself

To recreate 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.

```
ADDLIBLE 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 Figure 14 on page 110.

4.2.7.8 Files used in the condition handling example

For completeness, here is the source of the files used in the conditional error handling examples. The STOCK and STOCKW files are used in programs E01REG.

Physical file description: STOCK

```
 *****************************************************************
 *   STOCK  from ILESRC in RPGISCOOL
 *   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

This is a sample content of the STOCK file, which is used with our running example:

```
Stock   Item              Quantity
number  number            onhand
00001   00001                   12
```

Display file STOCKW

```
 *****************************************************************
  *   STOCKW from ILESRC in RPGISCOOL
  *   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  80                              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 is used in the E01ITEMS program:

```
 *****************************************************************
  *   ITEMS from ILESRC in RPGISCOOL
  *   Item master file
 *****************************************************************
A                                      UNIQUE
A          R ITEMSR
 *   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 using the following commands:

```
CRTPF FILE(RPGISCOOL/STOCK) SRCFILE(RPGISCOOL/ILESRC) SRCMBR(*FILE)
CRTPF FILE(RPGISCOOL/ITEMS) SRCFILE(RPGISCOOL/ILESRC) SRCMBR(*FILE)
CRTPRTF FILE(RPGISCOOL/REPORTS) SRCFILE(RPGISCOOL/ILESRC)
```

These files are used by the two programs described in 4.2.7.6, “Creating a program that uses the condition handler” on page 109. The STOCK file can contain data specified in “Physical file content: STOCK” on page 114. An example of the contents of file ITEMS is listed in “File ITEMS” on page 95.

4.3 Additional CL commands and useful ILE APIs

Other means for working with modules, service programs, and programs are covered in the next two sections at a high-level.

4.3.1 Additional CL commands

Other commands and parameters may be useful if working on a larger project. The Update Program (UPDPGM) and Update Service Program (UPDSRVPGM) commands make it possible to create a new 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 may be undesirable. For further information on updating programs and service programs, see *ILE Concepts,* SC41-5606.

The OPTION(*UNRSLVREF) parameter in the CRTPGM and CRTSRVPGM commands (also in the UPDPGM and UPDSRVPGM commands) allows unresolved references (imports) in the program or service program. This can be useful if testing a larger application. Some modules can refer to subprocedures that do not exist yet, but their interface is already known.

For further information on updating programs and service programs and unresolved references, see *ILE Concepts,* SC41-5606.

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 could use some API programs that are available for ILE. They are described in *OS/400 Program and CL Command APIs V4R4*, SC41-5870. The following API programs may be especially useful:

- The List Module Information (QBNLMODI) API lists information about modules. The information is placed in a user space specified by you. This API is similar to the Display Module (DSPMOD) command. You can use the QBNLMODI API to:
  – List the symbols defined that can be exported to other modules
  – List the symbols that are defined external 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:
  - List modules bound into a service program
  - List service programs bound to a service program
  - List data items exported to the activation group
  - List data item imports that are resolved by weak exports that were exported to the activation group
  - List copyrights of a service program
  - List procedure export information of a service program
  - List data export information of a service program
  - List signatures of a service program
- The List ILE Program Information (QBNLPGMI) API gives information about ILE programs, similar to the Display Program (DSPPGM) command. The information is placed in a user space specified by you. You can use the QBNLPGMI API to:
  - List modules bound into an ILE program
  - List service programs bound to an ILE program
  - List data items exported to the activation group
  - List data item imports that are resolved by weak exports that were exported to the activation group
  - List copyrights of an ILE program

You can, for example, list signatures of service programs bound in a program 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 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 retrieved by these APIs is organized hierarchically.

4.4 More information about ILE and shared open data paths

For additional information on shared open data paths, refer to the AS/400 Information Center found at `http://www.as400.ibm.com/infocenter`

Once inside the Information Center, select **Database and File Systems** -> **Data Management** or **DB2 UDB for AS/400 Database Programming**.

You can also consult these publications for more information:

- There is essential information on ILE in *ILE Concepts,* SC41-5606, and *ILE RPG for AS/400 Programmer's Guide*, SC09-2507.
- *ILE Application Development Example V4R1*, SC41-5602, shows how to use ILE concepts in practice.
- Another good source of information on ILE is *Moving to Integrated Language Environment for RPG IV*, GG24-4358.

- ILE APIs, such as CEETREC for ending an activation group or CEEHDLR for registration a condition handler program and many more, are described in *OS/400 Integrated Language Environment (ILE) CEE APIs V4R4,* SC41-5861.
- *OS/400 Program and CL Command APIs V4R4*, SC41-5870, contains APIs associated with ILE objects, especially those for retrieving information on modules, service programs, and programs in user space.

Chapter 5. Exploring new ways to exploit your AS/400 system

This chapter focuses on demonstrating how you can enhance your applications by accessing the wealth of Application Programming Interfaces (APIs) that are shipped with every AS/400 system. Perhaps we should point out at this stage that we are using the term "API" here in its most liberal and literal sense. That is, we are including in this category any callable interface on the system.

While some of these APIs have been available to RPG programmers for many years, many others only became accessible with the advent of prototyped calls in V3R2 and V3R6. Hopefully we can help you to uncover a couple of "gems" that may have escaped your notice. For example, did you know that your RPG IV programs can:

- Access the C library math functions?
- Drive TCP/IP Sockets applications?
- Directly process "PC type" files in the Integrated File System?
- Drive Web pages through CGI interfaces?
- Use the Lotus Domino HiTest APIs?

Be assured that if and when the required APIs are available on the AS/400 system, RPG IV will be there ready and able to exploit them!

We start by looking at a case study in 5.1, "Exploiting the C function library: A case study" on page 119. Until V3R2 and V3R6, these were the exclusive preserve of the C programmer.

5.1 Exploiting the C function library: A case study

In this section, we discuss the steps required to interface RPG IV programs to the C function library. We start with a simple example that uses one of the math functions to demonstrate the basics of interfacing to C routines.

The second example demonstrates interfacing to C routines that use pointers and null terminated strings. It is important to understand how RPG interfaces to such routines because they are common in the C world.

The third example demonstrates the use of more complex functions, in our case, the C sort and search routines. These can be invaluable when handling large arrays and Multiple Occurrence Data Structures (MODS).

Using CRTBNDRPG

The instructions for compiling the examples in this section suggest that you use the Create Bound RPG Program (CRTBNDRPG) command. We have done this deliberately to help dispel the myth that binding to service programs (the mechanism used by IBM to package the C functions) can only be done if you use the Create RPG Module (CRTRPGMOD) and Create Program (CRTPGM) commands. We are confident that if you already understand how to build programs by using a combination of the CRTRPGMOD and CRTPGM commands, you can adapt the examples to meet your needs.

5.1.1 First things first

The simplest, and sometimes only, way to interface to C functions is by prototyping the interface. The concept of prototyping is covered in more detail in 3.6.1, “The power of prototyping” on page 50. In this section, we reference those features of prototyping essential or useful in working with C functions.

Before we look at the work examples, we also need to understand the different ways in the which the C and RPG languages define data. Table 5 illustrates the basic C data types and their corresponding RPG types. Note that some of the RPG data types in this table are some of the newer data types for integers, which map best to the C numeric data types. Signed and unsigned integer data (types I and U, respectively) were introduced into RPG with V3R6 and V3R2. The Floating point data type (F) was introduced in V3R7. The pointer data type (*) has been in RPG IV since the beginning in V3R1.

Table 5. Correspondence of C and RPG IV data types

C	RPG IV	Notes
int, long	10I 0 Value	See note 1.
unsigned int	10U 0 Value	See note 1.
double	8F Value	See note 1.
char	10U 0 Value	See note 2.
short	10I 0 Value	See note 2.
int *	10I 0	See note 3.
unsigned *	10U 0	See note 3.
double *	8F	See note 3.
char *	* Value	Options(*String) can also be used; see note 4.
void *	* Value	See note 3.
(*)	* Value ProcPtr	See note 5.

The following notes correspond to the references in Table 5:

1. The normal method of parameter passing in RPG IV is known as "by reference". This means that a pointer to the data item is passed to the called routine, not the actual data itself. C, on the other hand, passes parameters "by value", which is the data itself is passed to the caller. In order for RPG to correctly pass such parameters, the keyword VALUE must be coded on the prototype. See also note 3.
2. In theory, *char* (which is a single byte character) should be represented in RPG IV as 1A and *short* (which is a short integer) as 5I 0. However, in practice, it is usually necessary to code such parameters as 10U 0 for *char* and 10I 0 for *short* since the C compiler lengthens these parameter types. If you are only using standard C library routines, this is unlikely to affect you since the libraries rarely use these data types. However, if you have some "home grown" C routines, you may run into this problem.

3. C classifies its pointers by the type of data to which it "points". Therefore, *int ** represents a pointer to an integer data item. RPG IV does not differentiate pointers this way so all RPG data pointers are equivalent to the C *void ** definition. As noted above, C expects parameters to be passed by value. However, passing a field "by reference" and passing a pointer to that field "by value" are equivalent. Since passing a parameter by field name rather than by using %Addr(fieldname) is somewhat more intuitive to RPG programmers. We used this notation in the table.

   This is demonstrated in the following example. CProto1 and CProto2 both reference the same C function (Cfunction), which takes a single parameter, an *int*. The CALLP operations in the example illustrate the differences in invocation.

   Note: Both of the following CALLP operations successfully call the same function passing the same data.

```
D CProto1         Pr                  ExtProc('Cfunction')
D  PtrToInt                       *   Value

D CProto2         Pr                  ExtProc('Cfunction')
D                               10I 0

D MyParm1         S             10I 0

C                   CallP     CProto1(%Addr(MyParm1))

C                   CallP     CProto2(MyParm1)
```

   The following variation illustrates the additional flexibility gained by adding the CONST keyword. The CONST keyword tells the compiler to allow a constant value or a variable with a different format (in terms of numeric type, length, or decimal positions) to be passed on the call. The CONST keyword is discussed in 3.6.1, “The power of prototyping” on page 50. Now, not only can MyParm1 be used, but MyParm2 (a packed field) and the literal 25 can also be used. The compiler generates the additional logic to convert the field to the correct format (if required) and pass it to the C function.

```
D CProto3         Pr                  ExtProc('Cfunction')
D                               10I 0 Const

D MyParm1         S             10I 0

D MyParm2         S              5P 0

C                   CallP     CProto3(MyParm1)
C                   CallP     CProto3(MyParm2)
C                   CallP     CProto3(25)
```

4. C does not really have the concept of a fixed-length character string in the way that RPG does. Most C functions expect strings to be variable in length and terminated by a null character (hex ’00’). RPG IV prototypes support the OPTIONS(*STRING) parameter to simplify the programmer’s life when interfacing to such routines. Using this option allows the programmer to specify either a field name or a pointer for the parameter. If a field name is used, the compiler generates code to move the field to a temporary area, add the null terminator, and then pass the address of this temporary area as the parameter.

Note: The *STRING option is available in RPG compilers on systems V3R7 or later. If you are on V3R2 or V3R6, you need to add the null terminator yourself. There is an example of doing this later in 5.1.3.1, “Defining the prototype” on page 125.

5. The keyword PROCPTR is used to qualify an RPG pointer (*) definition when the parameter in question is a procedure pointer. These are required by some C functions. For an example, see 5.1.4, “Searching and sorting: bsearch and qsort” on page 128.

5.1.2 Simple math functions

While RPG IV provides the basic math functions (Add, Subtract, Multiply, Divide) together with roots and exponents, it does not provide the higher level math functions such as Sine, Cosine, and Tangent. Admittedly, these are not functions that the average RPG programmer needs every day. But, when they are needed, "faking them out" in RPG is a real problem. Not any more!

The sample program CMATH uses the C Cosine function *cos* to facilitate the calculation of the length of the third side of a triangle given the length of two of the sides and the angle between them.

5.1.2.1 Defining the prototype

The C definition of the Cosine function is:

double cos(double)

This is translated in RPG IV to the following prototype:

```
  * Prototype for cos function
 D Cosine          Pr             8F   ExtProc('cos')
 D  Double                        8F   Value
```

The first line defines the name by which the RPG program refers to the function (Cosine) and also identifies the "real" procedure name of the function in the C library (cos). The entry 8F identifies the size and data type (double) of the value returned by the function.

It is essential to enter the name of the function in the correct case. Unlike regular RPG IV names, the names of procedures are case-sensitive. If it were mistakenly entered as "Cos," the CRTPGM step (or the bind step of CRTBNDRPG process) would fail since the system would be unable to locate the procedure.

The second line defines the parameter that is to be passed to the function. Note the use of the keyword VALUE, which, as noted earlier, is the default parameter passing method for C functions. The parameter name (Double) is ignored by the compiler and could have been left blank. We have chosen to adopt the convention of using a name that identifies the type of parameter expected.

5.1.2.2 Invoking the function

This couldn’t be simpler. You use the RPG name that was given on the prototype, just the same as you would if you were calling a subprocedure that you wrote yourself.

This is the part of the code where the Cosine function is invoked:

```
  * Calculate length of side 3
 C                   Eval(H)   Length3 = ( (Length1 ** 2) +
 C                                         (Length2 ** 2) -
 C                                         (2 * Length1 * Length2 *
 C                                         Cosine(Radians))) ** 0.5
```

5.1.2.3 Compiling the program: The magic of QC2LE

If you were to try to compile a program which included the prototype for cos simply by using PDM option 14 (CRTBNDRPG) and accept all of the defaults, the compile would fail. A study of the job log would reveal that it had failed because the system binder (which is automatically invoked following a successful RPG compile) was unable to resolve the symbol cos. In other words, the compiler does not know where to find the procedure that implements the function.

We could spend hours studying all of the service programs on the system and identifying which ones contained the required procedure. Then, we could bind it to our program. Luckily there is an easier way.

In order for the C compiler to find its library routines, IBM supplies a Binding Directory called QC2LE, which identifies the majority of service programs likely to be needed by a C program. Although intended for use by the C compiler, it can also be used by RPG programmers seeking to access C library functions.

One way of specifying this Binding Directory is through the BNDDIR parameter of the CRTBNDRPG command:

```
CRTBNDRPG PGM(RPGISCOOL/CMATH) SRCFILE(RPGISCOOL/QRPGLESRC) DFTACTGRP(*NO)
BNDDIR(QC2LE)
```

This can be simplified by taking advantage of one of the enhancements made to RPG IV in release V4R2. This provides the ability to embed compiler directives directly on the H specification. In our example, we have specified both the Default Activation Group parameter and the Binding Directory information. This allows the CRTBNDRPG command to be used with its defaults since the embedded values will override those specified on the command.

Here is the RPG IV code to achieve this:

```
 H BndDir('QC2LE')  DftActGrp(*NO)
```

5.1.2.4 Sample CMATH program

Here is the complete source code for our sample program:

```
  * Filename: RPGISCOOL/CFUNCTSRC(CMATH)

 H BndDir('QC2LE')  DftActGrp(*NO)

  * Prototype for cos function
 D Cosine          Pr             8F   ExtProc('cos')
 D  double                        8F   Value

 D PI              C                    3.14159

 D Msg1            C                    'Length of side 1'
 D Msg2            C                    'Length of side 2'
 D Msg3            C                    'Angle between sides'
 D Msg4            C                    'Length of side 3 is '

 D Length1         S              3P 0 Inz
 D Length2         S              3P 0 Inz
```

```
D Length3         S              7P 2 Inz
D Angle           S              3P 0 Inz
D Radians         S              8F   Inz
D Result          S             30

 * Ask user for lengths of sides and angle between them
C     Msg1          Dsply                   Length1
C     Msg2          Dsply                   Length2
C     Msg3          Dsply                   Angle

 * Convert Angle to Radians for use by Cos function
C                   Eval      Radians = (PI * Angle) / 180

 * Calculate length of side 3
C                   Eval(H)   Length3 = ( (Length1 ** 2) +
C                                         (Length2 ** 2) -
C                                         (2 * Length1 * Length2 *
C                                         Cosine(Radians))) ** 0.5

 * Edit length and create message to display result to user
C                   Eval      Result = Msg4 + %Trim(%EditC(Length3:'1'))

C     Result        Dsply
C                   Eval      *InLR = *On
```

Try it yourself

If you are using V4R2 or later, you can recreate the sample program using the following command:

```
CRTBNDRPG PGM(RPGISCOOL/CMATH) SRCFILE(RPGISCOOL/CFUNCTSRC)
```

If you are using a release prior to V4R2, you need to remove the H spec from the source and use the following command:

```
CRTBNDRPG PGM(RPGISCOOL/CMATH) SRCFILE(RPGISCOOL/CFUNCTSRC) DFTACTGRP(*NO)
BNDDIR(QC2LE)
```

To test the program, call it. When prompted, enter `3` and `4` for the lengths of sides 1 and 2, and `90` for the angle between them. The result should look like this:

```
DSPLY  Length of side 1      0
3
DSPLY  Length of side 2      0
4
DSPLY  Angle between sides      0
90
DSPLY  Length of side 3 is 5.00
```

5.1.3 String functions

C provides a number of string functions that can be useful to an RPG programmer. The example we have chosen to illustrate is *strtok.* This function can be used to break up a string into a series of "tokens". One use for this is to break up a description into its component words. We may want to do this so that we can eliminate multiple embedded blanks or certain special characters from the text, or simply to count the number of words used.

strtok allows you to specify which characters are to be considered the "delimiter" for these tokens. By this, we mean the individual characters (such as space, period, and comma) that mark the boundaries between tokens. The function also allows you to change the group of characters that represents the delimiter on each call to the function.

In our example, we use the colon as the delimiter for the first token, and a comma, period, semi-colon, colon, and space for all subsequent tokens. This means that if we input the string to our program:

"First token: Second, Third Last"

strtok would return "First Token", followed by "Second", then "Third", and finally "Last". Even though there is a space between the words "First" and "token", it will not be treated as a delimiter since the only valid delimiter at this point is a colon. Note also that the delimiter itself is not returned as part of the string and that multiple delimiters are ignored. This can be seen in the previous example where the token "Second" was delimited by a comma and then a space.

5.1.3.1 Defining the prototype

Now that we have an idea of how the function works, let's put it to use. The C definition of the function is:

*char * strtok (char * string , const char * delimiters)*

There are a number of different ways in which this can be represented as an RPG IV prototype. This is a simple version. However, it is not the one that we used in the example, for reasons that will become apparent in a moment.

```
D GetToken        Pr               *   ExtProc('strtok')
D  String@                         *   Value
D  Delimiters@                     *   Value
```

As before, the first line identifies the name by which the RPG program will invoke the function (GetToken), its real name (strtok), and the data type of the function's return value (*).

The second line identifies the first parameter, the input string to be tokenized. The C definition of this parameter (*char * string*) tells us that the parameter is a pointer to a character string (*). As with all C parameters, it is to be passed by the value (Value).

The third line identifies the second parameter, which contains the delimiters to be used. Its definition is identical to that of the first parameter.

As noted earlier, C strings are normally null (X'00') terminated, and *strtok* certainly expects this. RPG character fields, on the other hand, are not. Because of this, the RPG programmer must take steps to ensure that the string is null terminated before passing it. One way of doing this would be as follows:

```
C                   Eval      Temp1 = String + X'00'
C                   Eval      Temp2 = Delims + X'00'
C                   Eval      Token@ = GetToken(%Addr(Temp1):%Addr(Temp2))
```

Luckily, the people who designed the RPG IV compiler gave us the ability to specify OPTIONS(*STRING) on the prototype. When we use this option, the compiler checks if the parameter being passed is a pointer, or a character field. If it is a field, the compiler moves the contents of the field to a temporary area, null terminates it, and then passes the address of this temporary area as the parameter. This is the version that we used in our example.

Note: The *STRING option was added to the compiler in V3R7.

This is the modified version of the prototype:

```
D GetToken        Pr              *   ExtProc('strtok')
D  String@                        *   Value Options(*String)
D  Delimiters@                    *   Value Options(*String)
```

This allows the function to be invoked like this:

```
C                   Eval      Token@ = GetToken(String:Delims)
```

5.1.3.2 Invoking the function

The *strtok* function can be invoked in two different ways:

- If the first parameter is a valid string pointer, the function returns a pointer to the first token found in that string, or a null pointer if no valid token is found. This occurs, for example, if the delimiters included the space character and the input string consisted of nothing but spaces.

  In our example, the first invocation looks like this:

  ```
  C                   Eval      Token@ = GetToken( InpString : ':' )
  ```

  The field InpString is passed as the first parameter, and the literal `':'` is passed as the delimiter.

- If the second parameter is a null pointer, *strtok* returns a pointer to the next token in the original string. As before, a null pointer is returned when there are no (more) valid tokens.

  This is the second invocation in the sample program:

  ```
  C                   Eval      Token@ = GetToken( *Null : Delimiters )
  ```

  In this case, we pass a null pointer (*Null) as the first parameter to instruct *strtok* to continue to process the string passed on the first invocation. Note, however, that the second parameter now references the constant Delimiters that contain the characters comma, period, semi-colon, colon, and space. This allows us to demonstrate that the delimiters can be different on each invocation of the function.

5.1.3.3 Additional RPG IV C string support

We should note at this point that *strtok,* like many other C functions, returns a pointer to a string and not the string itself as an RPG program normally would. Not only that, but the string referenced by the pointer will be null terminated. This is not what our RPG code is expecting at all!

We can define a field as being BASED on the pointer returned by the function, and then use a substring operation to get rid of the null terminator. However, this would require extra work.

A far better approach is to take advantage of RPG IV's %STR built-in function (added in V3R7), which is designed to operate with C-type strings. When used on the right-hand side of an expression, as in our example, this function returns the data referenced by the pointer supplied, up to but not including the first null character (x'00') found. This is how it looks in our example:

```
C                   Eval      Token = %Str(Token@)
```

We can also use %STR on the left-hand side of an expression. In this case, the function assigns the value of the right-hand side of the expression to the memory referenced by the pointer, and adds a null-terminating byte at the end. In this, it is similar to the function of the prototype option OPTIONS(*STRING) described in 5.1.3.1, “Defining the prototype” on page 125.

5.1.3.4 Sample program STRTOK

An exercise for you

Notice that some of the following variables have broken the style guide rule found in 2.1.3.3, “Avoid using special characters (for example, @, #, $) when naming items” on page 22.

Avoid the "@" symbol to indicate the variable is a pointer. A good approach is to use the "Hungarian Notation", where the first character of the variable name indicates the data type of the variable, for example, pCustNbr or cActStsCde.

Here is the complete source for the sample program:

```
  * Filename: RPGISCOOL/CFUNCTSRC(STRTOK)

 H  BndDir('QC2LE') DftActGrp(*No)

 D GetToken        Pr              *   ExtProc('strtok')
 D  String@                        *   Value Options(*String)
 D  Delimiters@                    *   Value Options(*String)

 D InpString       S             36A
 D Delimiters      S              5A   Inz(',.:; ')

 D Token           S             36A
 D Token@          S               *
 D TokenCount      S              3P 0
 D Message         S             52A

 D Exit            C                   'Exit'
 D AskForInput     C                   'Enter test data'
 D AllDone         C                   'All tokens processed'

  * Ask for initial input string

 C     AskForInput   Dsply                   InpString

 C                   DoW       InpString <> Exit

 C                   Eval      Token@ = GetToken( InpString : ':' )

 C                   If        Token@ <> *Null
 C                   Eval      Token = %Str(Token@)
 C                   Eval      Message = 'First Token is: ' + Token
 C                   Eval      TokenCount = 1
 C     Message       Dsply
 C                   EndIf

 C                   DoW       Token@ <> *Null

  * To extract second and subsequent tokens, use *Null for the string
  *   parameter so that strtok will use the current string
  *   A new set of delimiters is being used. They can be changed on each call.

 C                   Eval      Token@ = GetToken( *Null : Delimiters )

  * Continue to extract tokens until Token@ is null indicating end of string
 C                   If        Token@ <> *Null

  * If a token is found increment the count and format the message
 C                   Eval      TokenCount = TokenCount + 1
 C                   Eval      Token = %Str(Token@)
 C                   Eval      Message = 'Token ' +
 C                                       %Trim(%EditC(TokenCount:'1')) +
 C                                       ' is: ' + Token
 C     Message       Dsply
```

```
C                   Else
C     AllDone       Dsply
C                   EndIf

C                   EndDo

 *Request next input string
C     AskForInput   Dsply                   InpString

C                   EndDo

C                   Eval      *InLr = *On


 *Request next input string
C     AskForInput   Dsply                   InpString

C                   EndDo

C                   Eval      *InLr = *On
```

Try it yourself

If you are using V4R2 or later, you can recreate the sample program using the following command:

```
CRTBNDRPG PGM(RPGISCOOL/STRTOK) SRCFILE(RPGISCOOL/CFUNCTSRC)
```

If you are using a release prior to V4R2, you need to remove the H spec from the source and use the following command:

```
CRTBNDRPG PGM(RPGISCOOL/STRTOK) SRCFILE(RPGISCOOL/CFUNCTSRC) DFTACTGRP(*NO)
BNDDIR(QC2LE)
```

To test the program, call it. When prompted, enter `Token One: Two, Three Four.End` as the test data. The result should look like this:

```
DSPLY  First Token is: Token One
DSPLY  Token 2 is: Two
DSPLY  Token 3 is: Three
DSPLY  Token 4 is: Four
DSPLY  Token 5 is: End
DSPLY  All tokens processed
```

5.1.4 Searching and sorting: bsearch and qsort

While RPG IV provides the LOOKUP and SORTA operations to handle arrays, they have a number of shortcomings. Luckily there are two C functions: qsort, which sorts an area of memory, and bsearch, which searches. This example demonstrates their use.

5.1.4.1 What’s wrong with LOOKUP and SORTA?

RPG programmers have been using these functions for years, so what’s wrong with them? There are quite a few things as it turns out, for example:

- They operate directly only on arrays and not on Multiple Occurrence Data Structures (MODS).
- They operate on the entire array as defined and not on the actual number of active entries. That is, if the array is not full, the programmer has to make allowance for the "empty" entries at the end of the array.

- Because they attempt to operate on the defined maximum number of elements in the array, they are not suitable for use with BASED arrays (they are "based" on a pointer).

  The storage for these could be allocated by the programmer using the ALLOC and REALLOC operation codes. They are often used to allow the programmer to "grow" the amount of storage allocated to an array as records are added. This avoids the need to reserve a large amount of storage "just in case". Storage is only used as and when needed.
- They are somewhat inflexible in terms of the sort/search collating sequence. For example, an array filled with customer names may not sequence "ACME" and "Acme" together since the sort is based on the EBCDIIC sequence. In this case, the characters "C" and "c" are not equivalent. The collating sequence of the program could be changed so that SORTA would sequence them together. However, this would also have the effect of changing the behavior of *all* comparison operations in the program, not just those related to the array in question.

5.1.4.2 Solving the problems

Can qsort and bsearch address all of these problems? Yes they can. Let's look at each of the points raised in the previous section:

- They can operate on any contiguous area of memory including arrays and MODS.
- They operate on the number of elements specified by the programmer, not the maximum capacity. They can even start the sort at an element other than the first, much like LOOKUP. However, unlike LOOKUP, they can be told to stop after a specific number of elements.
- They are suitable for use with BASED arrays and MODS.
- They are flexible. How flexible? As flexible as the RPG language. You determine the collating sequence with your own RPG code! In the case of the example given above, if your code converted the fields to upper case before comparing them, "Acme" would have become "ACME" and the sequencing would work as desired.

5.1.4.3 How do qsort and bsearch work?

Before we discuss the prototypes, a brief description of the operation of these routines may be necessary. We'll start with qsort since everything we say about it is also applicable to bsearch.

qsort operates by selecting a pair of elements from the array (or MODS) and passing them to a comparison procedure that you write in RPG. Such procedures are often referred to as *call back procedures* because the procedure that you call directly (qsort) will call "back" into your code to make sequencing decisons.

Your code examines the contents of the elements and determines the order in which they should be sorted:

- If element 1 should follow element 2, your procedure should indicate a "High" condition by returning a value of 1.
- If element 1 should precede element 2, you should indicate a "Low" condition by returning a value of -1.

- If the two elements are equal, you indicate that condition by returning a value of zero.

qsort looks at the High/Low/Equal value that you return and then reorders the elements as necessary. It then picks another pair and, again, calls your procedure to identify how they should be ordered. Once *qsort* determines that all elements are in the correct sequence, it returns control to your main line code.

bsearch differs only in that one of the elements it passes to your code is the search key that your mainline code supplied. The other is the current candidate entry. Again, your comparison routine returns High/Low/Equal and *bsearch* uses this to narrow its search range. Eventually, bsearch either finds the correct entry, in which case it returns a pointer to the item, or it returns a null pointer to indicate that no match was found.

Note: The type of search used by *bsearch* is sometimes referred to as a "binary chop". This is significantly faster than the method employed by LOOKUP, but will *only* work on sequenced data.

5.1.4.4 Defining the prototypes: qsort

The C definition of the *qsort* function is:

*void qsort(void *base, size_t num, size_t width,*

*int(*compare)(const void *key, const void *element))*

While this may look a lot more complex than our previous examples, it really is not. Here is the RPG IV translation of the main part:

```
D SortIt          PR                  ExtProc('qsort')
D  DataStart                      *   Value
D  Elements                    10U 0 Value
D  Size                        10U 0 Value
D  CompFunc                       *   ProcPtr Value
```

As before, the first line defines the name by which the RPG program refers to the function (SortIt) and also identifies the "real" procedure name of the function (qsort).

In the C definition, the function name is preceded by the word *void* indicating that this function does not return a value. We indicate this on the prototype by leaving the size and data type areas of the PR line blank. This means that we cannot use the function in an expression, but instead must use CALLP (Call with Prototype) to invoke the function.

The second line defines the first parameter as a pointer (*) passed by VALUE and should contain the address of the first byte of the array or MODS to be sorted. This corresponds to the entry *void *base* in the C definition. In this case the word *void* indicates that the pointer can "point" to any type of data. Since all data pointers in RPG are of this type, nothing special is needed in the RPG prototype.

The second parameter is an unsigned integer (10U 0) passed by VALUE, which contains a count of the number of elements to be sorted. This corresponds to *size_t num* in the C definition. *size_t* is defined by IBM on the AS/400 system as being equivalent to an *int.*

The third defines the size of each element in the array in bytes. The mapping of the C to the RPG IV definition should be clear on this one.

The fourth parameter is the tough one. If you look at the RPG IV definition, you can see that it is a pointer (*) passed by value. The additional keyword, PROCPTR, indicates that this is a procedure pointer. It is used to enable *qsort* to call an RPG IV subprocedure, which will be responsible for making the actual sequencing decisions. It corresponds to *(*compare)* in the C definition.

At this point, you may be wondering what the rest of the C definition is all about. Simply put, it defines the parameters that *qsort* passes to your RPG IV comparison procedure. It also defines the format of the High/Low/Equal return value that *qsort* expects you to pass back to it.

The procedure interface in our RPG procedure looks like this:

```
D SeqArray        PI            10 I 0
D  Element1@                       *   Value
D  Element2@                       *   Value
```

As you can see, the procedure is defined as returning a 4-byte integer. This may strike you as a little odd considering that only the values 1, 0, and -1 are returned. This is normal for a C function. The original C definition specifies this requirement through the keyword *int* in the original C definition:

*int(*compare)(const void *key, const void *element))*

The two parameters passed to our procedure (Element1@ and Element2@) are pointers to the pair of items that *qsort* wants us to compare. They equate to the entries *const void *key* and *const void *element* respectively. Since these are pointers to the data items, and not the items themselves, we must associate them with fields before we can compare the data. This is achieved by using the keyword BASED on the field definition. Also notice that we used the keyword LIKE to define the format of the field. This ensures that the definition matches that of the original array elements.

```
D Element1        S                    Like(Array) Based(Element1@)
D Element2        S                    Like(Array) Based(Element2@)
```

The effect of this is that when our procedure is called, we have immediate access to the two items, just as if they had been directly passed as parameters. We then compare the items, and return the appropriate value to *qsort.*

```
C                   Select
C                   When      Element1 < Element2
C                   Return    Low
C                   When      Element1 > Element2
C                   Return    High
C                   Other
C                   Return    Equal
C                   EndSl
```

It is worth noting here that passing a pointer to a data item is the normal method of passing parameters on the AS/400 system. This is known as passing "by reference". That is to say, a reference (pointer) to the data is passed rather than the actual data itself. See 3.6.1, “The power of prototyping” on page 50, for more information.

When you perform a conventional CALL or CALLB, the compiler effectively inserts the BASED keyword "under the covers", and you don’t have to concern yourself with it.

We chose, in our example, to use the BASED method so that you can appreciate the mechanics of the process. There will be occasions when you need to use this method, for example, when the function you are calling returns an array of pointers. If you find this difficult to understand, look at 5.1.4.8, “An alternative approach” on page 136, where a more conventional RPG approach is used to achieve the same effect. The result is a little less flexible but perhaps more "RPG like".

5.1.4.5 Invoking qsort

Since *qsort* does not return a value, it has to be invoked by the CALLP (CALL with Prototype) op-code. This is the actual code:

```
 C                   CallP     SortIt( Array@ : Count :
 C                                     %Size(Array) : %PAddr('SEQARRAY'))
```

Note that the name of the sequencing procedure SeqArray is specified in uppercase (’SEQARRAY’). This is because by default, RPG IV converts all names to uppercase. If for some reason you wish to use a different name, for example, to retain the mixed-case name used in the source, you can do so by using the EXTPROC keyword on the prototype. See *ILE RPG for AS/400 Reference*, SC09-2508, for more information.

5.1.4.6 Using bsearch

As noted earlier, *bsearch* is similar to *qsort*. The major difference is that *bsearch* requires an additional parameter (LookFor), which supplies the value for which to search.

As you can see from the prototype, this is in the form of a pointer, passed as usual by value. Here is the prototype:

```
D FindIt          PR              *   ExtProc('bsearch')
 D  LookFor                        *   Value
 D  DataStart                      *   Value
 D  Elements                     10U 0 Value
 D  Size                         10U 0 Value
 D  CompFunc                       *   ProcPtr Value
```

The RPG IV comparison procedure (CheckMatch) operates the same way as the earlier SeqArray procedure, returning a High/Low/Equal value to the *bsearch* function. Because *bsearch* is a function and returns a value, it can be invoked directly in an EVAL operation:

```
 C                   Eval      Entry@ = FindIt( %Addr(SearchFor) : Array@ :
 C                                              Count : %Size(Array) :
 C                                              %PAddr('CHECKMATCH') )
```

As you can see the returned value is assigned to the pointer Entry@, which is used as the base pointer for the field Entry. If *bsearch* found a match (of course it was the RPG IV procedure CHECKMATCH that really identified the match), Entry@ points to the correct array element that can be accessed through the field Entry. If no match is found, Entry@ is null. This is the signal for the program to add the entry to the array and re-sequence the array if necessary.

The following portion of the program demonstrates that Pointer arithmetic can be used to determine the actual array element (Elem) that the pointer is currently referencing:

```
C                   Eval      Elem = (( Entry@ - Array@ )
C                                      / %Size(Array)) + 1
```

This, in effect, renders the field Entry obsolete since it maps the same element as Array(Elem).

The technique demonstrated here to determine the actual element number can be useful in situations where there are multiple entries with the same value. This could occur for example, if you were to sort a Multiple Occurrence Data Structure (MODS) into Customer Name in City sequence. There may be multiple entries for the same city. In such cases, you cannot guarantee that the element identified by *bsearch* is, in fact, the first one of that value. If it is important that you process the first such element, you can do so by calculating the element number as shown and then "walk" backwards up the array until you have identified the first entry in the group.

5.1.4.7 The sample program

The program prompts the user to enter a search value. This can be any string up to 10 characters long. *bsearch* is then used to attempt to locate the string in the array. If the string is located, the program reports the element number at which it was found and prompts for another search string. If the string is not found, the program adds it to the end of the current array. If necessary, it then sorts the array using *qsort*.

Note that the array in this program uses dynamic memory, which is allocated by the ALLOC and REALLOC op-codes. This allows the array to grow in size as required.

An exercise for you

Notice that some of the following variables have broken the style guide rule found in 2.1.3.3, “Avoid using special characters (for example, @, #, $) when naming items” on page 22.

Avoid 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, for example, pCustNbr or cActStsCde.

```
H DftActGrp(*No) BndDir('QC2LE')

 /Copy CSortPr

D Array           S             10    Dim(100) Based(Array@)

D Entry           S             10    Based(Entry@)

D Count           S              3P 0 Inz
D ArrayMax        S              3P 0 Inz(10)
D Memory          S              5P 0
D Elem            S              3P 0

D SearchFor       S             10

D FoundMsg        S             24
```

```
D NotFoundMsg     C                   'No entry found - added to list'

 * Prototypes for compare routines for search and sort
D CheckMatch      Pr            10I 0
D  CheckFor@                      *   Value
D  Candidate@                     *   Value

D SeqArray        Pr            10I 0
D  Element1@                      *   Value
D  Element2@                      *   Value

 * Set up initial memory allocation and load a dummy entry
C                   Eval      Memory = %Size(Array) * ArrayMax
C                   Alloc     Memory        Array@
C                   Eval      Array(1) = 'first one'
C                   Eval      Count = Count + 1

 * Ask for value to search for and exit if 'quit'
C     'Search For?' Dsply                   SearchFor

C                   DoU       SearchFor = 'quit'

 * Call search routine - CHECKMATCH is the RPG procedure that will
 *   determine if a match has been found
C                   Eval      Entry@ = FindIt( %Addr(SearchFor) : Array@ :
C                                             Count : %Size(Array) :
C                                             %PAddr('CHECKMATCH') )

 * If a match is found the pointer will be set non null, so we will
 *   calculate which element it is and display the result
C                   If        Entry@ <> *Null

C                   Eval      Elem = (( Entry@ - Array@ )
C                                       / %Size(Array)) + 1
C                   Eval      FoundMsg = 'Found - Element # is ' +
C                                        %TrimL( %EditC(Elem : 'Z') )
C     FoundMsg      Dsply


C                   Else

 * If no match is found then we issue a message and add the entry to the
 *   end of the array after first increasing the storage if required
C     NotFoundMsg   Dsply

C                   Eval      Count = Count + 1

C                   If        Count > ArrayMax
C                   Eval      Memory = %Size(Array) * ArrayMax
C                   ReAlloc   Memory        Array@
C                   EndIf

C                   Eval      Array(Count) = SearchFor

 * Check to see if the array is still in sequence and sort if not
C                   If        SearchFor <= Array(Count -1)
C                   CallP     SortIt( Array@ : Count :
C                                     %Size(Array) : %PAddr('SEQARRAY'))
C                   EndIf

C                   EndIf

C     'Search For?' Dsply                   SearchFor

C                   EndDo

C                   Eval      *InLR = *On
 * Beginning of search procedure CheckMatch - this will be called by bsearch
P CheckMatch      B

D                 PI            10I 0
D  CheckFor@                      *   Value
D  Candidate@                     *   Value

D CheckFor        S                   Like(Array) Based(CheckFor@)
D Candidate       S                   Like(Array) Based(Candidate@)

C                   Select
```

```
C                   When      CheckFor < Candidate
C                   Return    Low
C                   When      CheckFor > Candidate
C                   Return    High
C                   Other
C                   Return    Equal
C                   EndSl

P CheckMatch      E

 * Beginning of sequencing procedure SeqArray - this will be called by qsort
P SeqArray        B

D                 PI            10I 0
D  Element1@                      *   Value
D  Element2@                      *   Value

D Element1        S                   Like(Array) Based(Element1@)
D Element2        S                   Like(Array) Based(Element2@)

C                   Select
C                   When      Element1 < Element2
C                   Return    Low
C                   When      Element1 > Element2
C                   Return    High
C                   Other
C                   Return    Equal
C                   EndSl

P SeqArray        E
```

Try it yourself

If you are using V4R2 or later, recreate the sample program using the following command:

```
CRTBNDRPG PGM(RPGISCOOL/SORTARRAY) SRCFILE(RPGISCOOL/CFUNCTSRC)
```

If you are using a release prior to V4R2, you need to remove the H spec from the source and use the following command:

```
CRTBNDRPG PGM(RPGISCOOL/SORTARRAY) SRCFILE(RPGISCOOL/CFUNCTSRC)
DFTACTGRP(*NO) BNDDIR(QC2LE)
```

To test the program, call it. When prompted, enter a word. In the following sample, we used "qsort". Continue to enter a word each time you are prompted. We used "bsearch", followed by "qsort" again. As you will see when you run the program, if the word was previously entered, the program informs you of its position in the array.

Your results should look something like this:

```
DSPLY  Enter string to search for  (quit to exit)
qsort
DSPLY  No entry found - added to list
DSPLY  Enter string to search for  (quit to exit)    qsort
bsearch
DSPLY  No entry found - added to list
DSPLY  Enter string to search for  (quit to exit)    bsearch
qsort
DSPLY  Found - Element # is 2
```

5.1.4.8 An alternative approach

As noted earlier, there is an alternative method that we can use for coding the parameters for our RPG sequencing and searching procedures. The changes to the SeqArray procedure is shown here. Similar changes would be made to CheckMatch:

```
D SeqArray        Pr            10I 0
D  Element1                           Like(Array)
D  Element2                           Like(Array)
 :
 :
P SeqArray        B
D                 PI            10I 0
D  Element1                           Like(Array)
D  Element2                           Like(Array)
 :
```

Note that the fields Element1 and Element2 are now defined directly in the procedure interface (PI). There is no need for a separate field definition and associated BASED keyword. The compiler does this for us.

The changes to the prototypes and procedure interfaces are the only ones required. The invocations of *qsort* and *bsearch* remain unchanged, as does the rest of the program logic.

We can use this approach because passing a field by reference results in a pointer to the data being passed to the called procedure. This is equivalent to passing a pointer to the field by value and using that pointer to base the field. See 3.6.1, “The power of prototyping” on page 50, for more information.

If you find this approach easier to understand, by all means, use it. You will find a version of the program that uses this method in source member SORTARRAY2. Other than the name of the source member, all of the other details for compiling and testing the program are the same.

A recommendation

Seeing these changes, you may be tempted to consider revising the original prototypes for *qsort* and *bsearch* to use the same approach. This would not be a good idea. If you were to do this, you would have to code a separate prototype for each different array or MODS that you want to sort or search to accommodate the differences in length and data type of the array. By using pointers, we create a more generic prototype that is able to handle any kind of structure.

5.2 Data queue APIs

Data queues are a type of system object that a user can create to establish communication between two high-level language (HLL) programs. One HLL program can send data to the data queue, and another HLL program can receive data from the data queue. The receiving program can wait for the data and accept it immediately or receive the data later.

The usability of data queues is not restricted only to the programs residing on the same AS/400 system. They can be used also to establish communications between PC clients and AS/400 server or between high-level language programs running on different AS/400 systems. Figure 21 shows the different communications possibilities by using data queues.

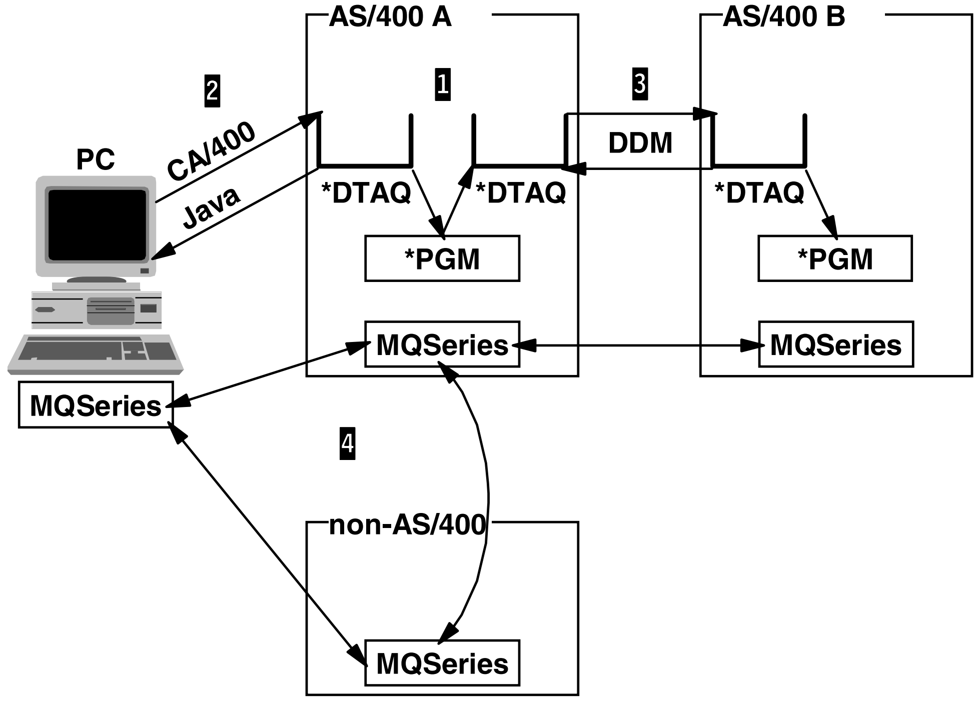


Figure 21. Data queue communications possibilities

Note the following points as they refer to Figure 21:

1 Communication between HLL programs residing on the same AS/400 system is the most common example of data queue implementation. We concentrate our on this discussion later.

2 Data queues are also an excellent way to communicate between the PC and AS/400 programs. Client Access/400 provides a complete Application Programming Interface (API), which allows client application to interact with AS/400 data queues. This interface allows the client application to create and delete data queues, as well as send and read free format messages from data queues.

The data queue interface uses an AS/400 host server to actually execute the data queue commands against the AS/400 objects. This server can be started either using the STRHOSTSVR SERVER(*DTAQ) CL command, or through data queue APIs. PC programs can be written in any language that provides a *DTAQ interface such as C, Delphi, Visual Basic, and VisualAge for RPG.

3 By creating a data queue with the type *DDM, we actually refer to a remote data queue on the target system. Using this type of data queue, programs on different AS/400 systems can efficiently exchange data.

4 If there is a need to exchange data with other non-AS/400 systems, either IBM or non-IBM, a separate product MQSeries can be used. This product enables applications to use message queuing to participate in message-driven processing. This allows applications to communicate with each other on the same or different platforms by using the appropriate message queuing software products. With MQSeries products, all applications use the same kind of messages, while communications protocols are hidden from the applications.

MQSeries is a robust solution not only for heterogeneous systems but also for communications between two HLL on the same AS/400 system. For more information on MQSeries, go to: `http://www.software.ibm.com/ts/mqseries/`

MQseries understands RPG IV!

Here is a quote from *MQSeries for AS/400 Application Programming Reference (RPG)*, SC33-1957:

"There are two approaches that can be taken when using the Message Queue Interface (MQI) from within an RPG program:

- Dynamic calls to the QMQM program interface. This method is available to OPM and ILE RPG programs.
- Bound Calls to the MQI procedures. This method is available only to ILE RPG programs.

Using bound calls is generally the preferred method, particularly when the program is making repeated calls to the MQI, as it requires less resource."

The advantages of using data queues are:

- Data queues are a fast means of asynchronous communication between two jobs. Using a data queue to send and receive data requires less system resource than using database files, message queues, or data areas.
- Using data queues frees a job from performing work. If the job is an interactive job, the data queue APIs can provide better response time and decrease the size of the interactive program. For example, several workstation users may enter a transaction that involves updating and adding to several files. In this case, the system can perform better if the interactive jobs submit the request for the transaction to a single batch processing job.
- More than one job can receive data from the same data queue. This is an advantage in certain applications where the number of entries to be processed is greater than one job can handle within the desired performance restraints. For example, if several printers are available to print orders, several interactive jobs can send requests to a single data queue. A separate job for each printer can receive data from the data queue in first-in-first-out (FIFO), last-in-first-out (LIFO), or keyed-queue order.
- Data queues have the ability to attach a sender ID to each message being placed on the queue. The sender ID, an attribute of the data queue that is established when the queue is created, contains the qualified job name and current user profile.
- When receiving data from a data queue, you can set a time-out so that the job waits until an entry arrives on the data queue.
- You can send to, receive from, and retrieve a description of a data queue in any HLL program. This is done by calling the API programs:
  - Send to a Data Queue (QSNDDTAQ)
  - Receive from Data Queue (QRCVDTAQ)
  - Retrieve Data Queue Message (QMHRDQM)
  - Clear Data Queue (QCLRDTAQ)
  - Retrieve Data Queue Description (QMHQRDQD) APIs

5.2.1 Creating and deleting data queues

Before using a data queue, you must first create it using the Create Data Queue (CRTDTAQ) command.

The following parameters are important when creating a data queue:

- **TYPE**

  Specifies the type of data queue to be created: a standard data queue (*STD) or a distributed data management (*DDM) data queue.

- **MAXLEN**

  Specifies the maximum length of the entry that is sent to the data queue. Valid values range from 1 through 64512.

- **FORCE**

  Specifies whether the data queue is forced to auxiliary storage when entries are sent or received for this data queue.

- **SEQ**

  Specifies the sequence in which entries are received from the data queue:

  - *FIFO: Data queue entries are received in a first-in first-out sequence
  - *LIFO: Data queue entries are received in a last-in first-out sequence
  - *KEYED: Data queue entries are received by key. A key is a prefix added to an entry by its sender. Additional parameter KEYLEN specifies the length of the key. Valid values range from 1 through 256.

- **SENDERID**

  Specifies a sender ID to be attached to each message sent to the Data Queue. The ID contains the job name and the sender's current user profile.

To delete data queue, use the Delete Data Queue (DLTDTAQ) command.

5.2.2 List of data queue APIs

The access to data queues is possible only through the group of APIs that can be used from any high-level language program, including RPG IV.

5.2.2.1 Send Data Queue (QSNDDTAQ) API

The Send Data Queue (QSNDDTAQ) API sends data to the specified data queue.

When an entry is sent to a standard data queue, the storage allocated for each entry is the value specified for the maximum entry length on the Create Data Queue (CRTDTAQ) command.

This API has required and optional parameters as shown in Table 6.

Table 6. Required parameter group for QSNDDTAQ

Number	Description	Use	Data type
1	Data queue name	Input	Char(10)
2	Library name	Input	Char(10)
3	Length of data to be sent to data queue	Input	Packed(5,0)
4	Data to be sent to data queue	Input	Char(*)

For the library name, you can use the special values *LIBL (library list) or *CURLIB (current library).

To improve data queue performance, the data queue APIs remember addressing information for the last data queues used. This occurs when a specific (not *LIBL or *CURLIB) value is provided for the library name.

Because the addressing information is saved, users of this API should be aware of the following case. The data queue may be moved to another library or renamed, and a new data queue is created with the same name and library as the data queue that was renamed or moved. In this case, the job continues to reference the original data queue, not the newly created data queue.

If the value specified for the data length is greater than the length specified by the maximum length (MAXLEN) parameter on the Create Data Queue (CRTDTAQ) command, an error occurs.

If the length of data field is larger than the data length value parameter, only the number of characters (beginning from the left) defined by the data length are sent to the data queue.

If the length of data field is smaller than the data length value parameter, unexpected results can occur.

Table 7 shows optional parameter group 1, which is used only to send data queue entries with key.

Table 7. Optional parameter group 1 for QSNDDTAQ

Number	Description	Use	Data type
5	Length of key data	Input	Packed(3,0)
6	Key data	Input	Char(*)

The maximum value for the key length is the value that is specified on the KEYLEN parameter on the Create Data Queue (CRTDTAQ) command.

The key data value must be at least as long as the value specified in the key length parameter. Otherwise, unexpected results can occur.

Table 8. Optional parameter group 2 for QSNDDTAQ

Number	Description	Use	Data type
7	Asynchronous request	Input	Char(10)

This parameter only applies to DDM data queues. It specifies whether the send data queue request to a DDM data queue should be processed asynchronously. Valid values are *YES and *NO.

5.2.2.2 Receive Data Queue (QRCVDTAQ) API

The Receive Data Queue (QRCVDTAQ) API receives data from the specified data queue. When more than one program has a receive pending on a data queue at one time, a data entry sent to the data queue is received by only one of the programs. The program with the highest run priority receives the entry.

This API has required and optional parameters as shown in Table 9.

Table 9. Required parameter group for QRCVDTAQ

Number	Description	Use	Data type
1	Data queue name	Input	Char(10)
2	Library name	Input	Char(10)
3	Length of data received from data queue	Output	Packed(5,0)
4	Data received from data queue	Output	Char(*)
5	Wait time	Input	Packed(5,0)

The same comments for the data queue name and the library given with QSNDDTAQ API are valid also for QRCVDTAQ.

The data length parameter contains the number of characters received from the data queue. If a time out occurs and no data is received from the data queue, this field is set to zero.

If the length of data field is larger than the size of the message received, only the number of characters (beginning from the left), as defined by the message received from the data queue, are changed.

If the length of a data field is smaller than the value specified for the MAXLEN parameter on the Create Data Queue (CRTDTAQ) command, and the actual length of this field is not specified in the size of data receiver parameter, unexpected results can occur.

The Wait parameter specifies the amount of time, in seconds, to wait if no entries exist on the data queue:

0	Continue processing immediately. If no entry exists, the call completes immediately with the length of the data parameter set to zero.
> 0	The number of seconds to wait. The maximum is 99999, which allows a wait time of approximately 28 hours.
< 0	Waits forever.

Table 10 shows optional parameter group 1, which is used to retrieve data queue entries by key, or if sender information is to be received.

Table 10. Optional parameter group 1 for QRCVDTAQ

Number	Description	Use	Data type
6	Key order	Input	Char(2)
7	Length of key data	Input	Packed(3,0)
8	Key data	I/O	Char(*)
9	Length of sender information	Input	Packed(3,0)
10	Sender information	Output	Char(*)

The Key order parameter defines the comparison criteria between the keys of messages on the data queue and the key data parameter. When the system searches for the requested key, the entries are searched in ascending order from the lowest value key to the highest value key until a match is found. If there are entries with duplicate keys, the entry that was put on the queue first is received. Valid values are:

GT	Greater than
LT	Less than
NE	Not equal
EQ	Equal
GE	Greater than or equal
LE	Less than or equal

The key length parameter specifies the length of the key. If this parameter is specified, it must be zero for non-keyed data queues. For keyed data queues, it must be equal to the length specified on the KEYLEN parameter on the Create Data Queue (CRTDTAQ) command.

The Key data parameter defines the key to be used for receiving a message from the data queue. The key of the received message is also returned in this field, because it may be different from the key specified to search for. For example, if the Key order parameter is GE (greater than or equal to), the key of the actually received record may be greater than the requested key.

The length of the sender information parameter defines the requested size of the sender information. The valid values are:

0	No sender information is returned.
8	Returns only the bytes returned and bytes available fields of the sender information.
> 8	Return as much sender information as the length allows.

Format of sender information

The format and content of the sender information returned is shown in Table 11.

Table 11. Format of sender information

From	To	Description	Type
1	4	Bytes returned	Packed(7,0)
5	8	Bytes available	Packed(7,0)
9	18	Job name	Char(10)
19	28	User profile name	Char(10)
29	34	Job number	Char(6)
35	44	Senders current user profile name	Char(10)

Table 12 shows optional parameter group 2, which is used to leave retrieved entry in the data queue, receive only a part of entry, or to handle errors.

Table 12. Optional parameter group 2 for QRCVDTAQ

Number	Description	Use	Data type
11	Remove message	Input	Char(10)
12	Size of data receiver variable	Input	Packed(5,0)
13	Error code	I/O	Char(*)

The Remove message parameter specifies whether the message is to be removed from the data queue when it is received. The valid values are:

***YES**	The message is removed from the data queue. This is the default value if this parameter is not specified.
***NO**	The message is not removed from the data queue.

The data receiver size parameter defines the size of the variable to contain the data received from the data queue. If a value of 0 is specified for this parameter, no data is returned. If a size greater than 0 is specified, the data is copied into the receiver up to the specified length. If the available data is longer than the length specified, it is truncated. If this parameter is not specified, the entire message is copied into the receiver variable.

Error code parameter

The error code parameter defines the variable-length data structure, which contains the information associated with an error condition. It is common to all of the system APIs and appears in many examples provided in this redbook. Therefore, it deserves more attention and detailed explanation.

The error code parameter can be defined in two different formats: ERRC0100 or ERRC0200. In our examples, we use only the format ERRC0100, which is simpler but still provides enough information for error handling. For this reason, we provide only the explanation of format ERRC0100. If you need additional error information, refer to the Section "Error Code Parameter" in the IBM manual *AS/400 System API Reference*, SC41-5801, for the explanation of the format ERRC0200.

Table 13 shows the content of the error information in the format ERRC0100. The error code structure for this format is provided in the QUSEC include member in the source file QRPGLESRC in the QSYSINC library.

Table 13. Error format ERRC0100

From	To	Description	Use	Type
1	4	Bytes provided	Input	Integer(10)
5	8	Bytes available	Output	Integer(10)
9	15	Exception ID	Output	Char(7)
16	16	Reserved	Output	Char(1)
17	*	Exception data	Output	Char(*)

Bytes provided is an input parameter, which controls whether an exception is returned to the application or the error code structure is filled in with the exception information. Consider these points:

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

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

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

5.2.2.3 Retrieve Data Queue Message (QMHRDQM) API

The Retrieve Data Queue Message (QMHRDQM) API retrieves one or more messages from a data queue.

The QMHRDQM API allows the retrieval of multiple messages per call. The message selection information parameter allows you to have some control over which messages are returned. The QMHRDQM API can be used to retrieve:

- The first or last message of a data queue
- All messages of a data queue
- Selected messages from a keyed data queue

The QMHRDQM API is similar in function to the QRCVDTAQ API. However, the QRCVDTAQ API removes the received message from the data queue. QMHRDQM API does not remove received messages.

Table 14 shows the parameters for this API. All of them are required.

Table 14. Required parameter group for QMHRDQM

Number	Description	Use	Data type
1	Receiver variable	Output	Char(*)
2	Length of receiver variable	Input	Integer(10)
3	Format name	Input	Char(8)
4	Qualified data queue name	Input	Char(20)
5	Message selection information	Input	Char(*)
6	Length of message selection information	Input	Integer(10)
7	Message selection information format name	Input	Char(8)
8	Error code	I/O	Char(*)

The receiver variable specifies the program variable that receives the information requested. You can specify the size of the area to be smaller than the format requested as long as you specify the length parameter correctly. As a result, the API returns only the data that the area can hold.

The Length of receiver variable parameter may be specified up to the size of the receiver variable specified in the program. If the length of receiver variable parameter specified is larger than the allocated size of the receiver variable specified in the program, the results are not predictable. The minimum length is 8 bytes.

The format name specifies the format of the data to be placed in the receiver variable. You must use the RDQM0100 format.

The qualified data queue name defines the data queue to be retrieved. The first 10 characters contain the data queue name, and the second 10 characters contain the data queue library name. You can use special values *CURLIB and *LIBL for the library name.

Message selection information identifies which message (or messages) you want to retrieve. The layout of this parameter is determined by the value of the message selection information format name.

The length of the message selection information parameter must be 8 bytes for RDQS0100 and 16 bytes plus the size of the key for RDQS0200.

For the format of the message selection information, you can choose between the following formats:

- **RDQS0100**: Format to select messages when using nonkeyed data queues.
- **RDQS0200**: Format to select messages when using keyed data queues.

The Error code parameter defines the structure in which to return error information. Refer to “Error code parameter” on page 143.

RDQM0100 format

Table 15 shows the fields returned in the RDQM0100 format of the receiver variable parameter.

Table 15. RDQM0100 format

From	To	Description	Type
1	4	Bytes returned	Integer(10)
5	8	Bytes available	Integer(10)
9	12	Number of messages returned	Integer(10)
13	16	Number of messages available	Integer(10)
17	20	Message key length returned	Integer(10)
21	24	Message key length available	Integer(10)
25	28	Message text length returned	Integer(10)
29	32	Message text length available	Integer(10)
33	36	Entry length returned	Integer(10)
37	40	Entry length available	Integer(10)
41	44	Offset to first message entry	Integer(10)
45	54	Actual data queue library	Char(10)
55	*	Reserved	Char(*)
These fields repeat for each message retrieved		Offset to next message entry	Integer(10)
		Message enqueue date and time	Char(8)
		Message key	Char(*)
		Message text	Char(*)
		Reserved	Char(*)

RDQS0100 format

Table 16 describes the RDQS0100 format of the message selection information parameter. This format is used with data queues when selection with keys is not necessary. This format cannot be used with keyed data queues.

Table 16. RDQS0100 format

From	To	Description	Type
1	1	Selection type	Char(1)
2	4	Reserved	Char(3)
5	8	Number of message text bytes to retrieve	Integer(10)

The valid values for the selection type in the format RDQS0100 are:

A	All messages are to be returned
F	The first message is to be returned
L	The last message is to be return

RDQS0200 format

Table 17 describes the RDQS0200 format of the message selection information parameter. This format is used to retrieve messages from data queues when selection with keys is necessary. When using this format, all messages satisfying the key search order are returned. The messages are returned in first-in first-out order.

Table 17. RDQS0200 format

From	To	Description	Type
1	1	Selection type	Char(1)
2	3	Key search order	Char(2)
4	4	Reserved	Char(1)
5	8	Number of message text bytes to retrieve	Integer(10)
9	12	Number of message key bytes to retrieve	Integer(10)
13	16	Length of key	Integer(10)
17	*	Key	Char(*)

The valid value for the selection type in the format RDQS0200 is:

K Messages meeting the key criteria are to be returned

The Key search order parameter specifies the comparison criteria between the message key specified in the RDQS0200 format and the actual keys of messages in the data queue. Valid values are:

GT	All messages with a key greater than the one specified in the key field are to be returned.
LT	All messages with a key less than the one specified in the key field are to be returned.
NE	All messages with a key not equal to the one specified in the key field are to be returned.

EQ All messages with a key equal to the one specified in the key field are to be returned.

GE All messages with a key greater than or equal to the one specified in the key field are to be returned.

LE All messages with a key less than or equal to the one specified in the key field are to be returned.

5.2.2.4 Retrieve Data Queue Description (QMHQRDQD) API

The Retrieve Data Queue Description (QMHQRDQD) API retrieves the description and attributes of a data queue. Consider, for example, the number of entries currently on the data queue, the text description of the data queue, whether the queue includes sender ID information, or whether the data queue is keyed.

Table 18 shows the parameters for this API. All of them are required.

Table 18. Required parameter group for QMHQRDQDM

Number	Description	Use	Data type
1	Receiver variable	Output	Char(*)
2	Length of receiver variable	Input	Integer(10)
3	Format name	Input	Char(8)
4	Qualified data queue name	Input	Char(20)

The parameters for this API are the same as the first four parameters of the QMHRDQM API, and are already described.

The valid format name for this API is RDQD0100.

RDQD0100 format

Table 19 shows the fields returned in the RDQM0100 format of the receiver variable parameter.

Table 19. RDQD0100 format

From	To	Description	Type
1	4	Bytes returned	Integer(10)
5	8	Bytes available	Integer(10)
9	12	Message length	Integer(10)
13	16	Key length	Integer(10)
17	17	Sequence	Char(1)
18	18	Include sender ID	Char(1)
19	19	Force indicator	Char(1)
20	69	Text description	Char(50)
70	72	Reserved	Char(3)
73	76	Number of messages	Integer(10)
77	80	Maximum number of messages	Integer(10)

The possible values returned for the Sequence parameter are:

F	First-in first-out
K	Keyed
L	Last-in first-out

The possible values returned for the Include sender ID parameter are:

Y	The sender ID is included when data is sent to the data queue.
N	The sender ID is not included when data is sent to the data queue.

The possible values returned for the Force indicator parameter are:

Y	The data queue is forced to auxiliary storage after entries are sent or received.
N	The data queue is not forced to auxiliary storage after entries are sent or received.

5.2.2.5 Clear Data Queue (QCLRDTAQ) API

The Clear Data Queue (QCLRDTAQ) API clears all data from the specified data queue or clears messages that match the key specification from a keyed data queue.

This API has required and optional parameters. Table 20 shows the required parameters.

Table 20. Required parameter group for QCLRDTAQ

Number	Description	Use	Data type
1	Data queue name	Input	Char(10)
2	Library name	Input	Char(10)

These parameters are already described with QSNDDTAQ API.

Table 21 shows the optional parameter group, which is used to handle key information or error data.

Table 21. Optional parameter group for QCLRDTAQ

Number	Description	Use	Data type
3	Key order	Input	Char(2)
4	Length of key data	Input	Packed(3,0)
5	Key data	Input	Char(*)
6	Error code	I/O	Char(*)

These parameters are already described with QRCVDTAQ API.

5.2.2.6 RPG IV prototype for data queue APIs

Source member DTAQPROTO contains prototypes for calling data queue APIs using an RPG IV program:

```
 * Filename DTAQPROTO from APISRC in RPGISCOOL
 *
 * Prototype for API QSNDDTAQ - Send To a Data Queue
D SndDtaQ         PR                  EXTPGM('QSNDDTAQ')
D  DataQueueNam                 10A   Const
D  DataQueueLib                 10A   Const
D  DataLength                    5P 0 Const
```

```
D  DataBuffer                32767A   Const Options(*Varsize)
 * Optional parameter group  (Keyed DTAQ)
D  KeyLength                     3P 0 Const Options(*Nopass)
D  KeyBuffer                   256A   Const Options(*Nopass : *Varsize)
D  AsyncRqs                     10A   Const Options(*Nopass : *Varsize)
 *
 * Prototype for API QRCVDTAQ - Received From a Data Queue
D RcvDtaQ         PR                  EXTPGM('QRCVDTAQ')
D  DataQueueNam                 10A   Const
D  DataQueueLib                 10A   Const
D  DataLength                    5P 0
D  DataBuffer                32767A         Options(*Varsize)
D  WaitTime                      5P 0 Const
 * Optional parameter group 1 (Keyed DTAQ)
D  KeyOrder                      2A   Const Options(*Nopass)
D  KeyLength                     3P 0 Const Options(*Nopass)
D  KeyBuffer                   256A         Options(*Nopass : *Varsize)
D  SndLength                     3P 0 Const Options(*Nopass)
D  SndBuffer                    44A         Options(*Nopass)
 * Optional parameter group 2
D  RemoveMsg                    10A   Const Options(*Nopass : *Omit)
D  RcvSize                       5P 0 Const Options(*Nopass : *Omit)
D  Error                     32767A         Options(*Nopass : *Varsize)
 *
 * Prototype for API QCLRDTAQ - Clear Data Queue
D ClrDtaQ         PR                  EXTPGM('QCLRDTAQ')
D  DataQueueNam                 10A   Const
D  DataQueueLib                 10A   Const
 * Optional parameter group
D  KeyOrder                      2A   Const Options(*Nopass)
D  KeyLength                     3P 0 Const Options(*Nopass : *Omit)
D  KeyBuffer                   256A         Options(*Nopass : *Varsize)
D  Error                     32767A         Options(*Nopass : *Varsize)
 *
 * Prototype for API QMHQRDQD - Retrieve Data Queue Description
D RcvDtaQDesc     PR                  EXTPGM('QMHQRDQD')
D  RcvVar                      100A         Options(*Varsize)
D  RcvLength                    10I 0 Const
D  FormatName                    8A   Const
D  DataQueueNamL                20A   Const
 *
 * Prototype for API QMHRDQM - Retrieve Data Queue Message
D RcvDtaQMsg      PR                  EXTPGM('QMHRDQM')
D  RcvVar                    32767A         Options(*Varsize)
D  LengthRcv                    10I 0 Const
D  FormatName                    8A   Const
D  DataQueueNamL                20A   Const
D  MsgSltInf                 32767A   Const Options(*Varsize)
D  MsgSltLength                 10I 0 Const
D  MsgSltFmtName                 8A   Const
D  Error                     32767A         Options(*Varsize)
```

As you can see, we use the `OPTIONS(*VARSIZE)` keyword for all the parameters that don't require a predefined length, and specify the maximum field length as the default. The keyword `OPTIONS(*NOPASS)` indicates optional parameters that aren't mandatory when using the prototype call command, although the APIs required all the parameters from an optional group to be specified (or not) together. For more information on those parameters, the `OPTION(*OMIT)`, and the `CONST` keywords, see 3.6.1, "The power of prototyping" on page 50.

5.2.3 Programming with data queue APIs

Data queue APIs are used in an RPG IV program the same way as other OPM APIs.

5.2.3.1 Data queue API example

To illustrate the use of data queue APIs, we use two programs as shown in Figure 22 on page 150.

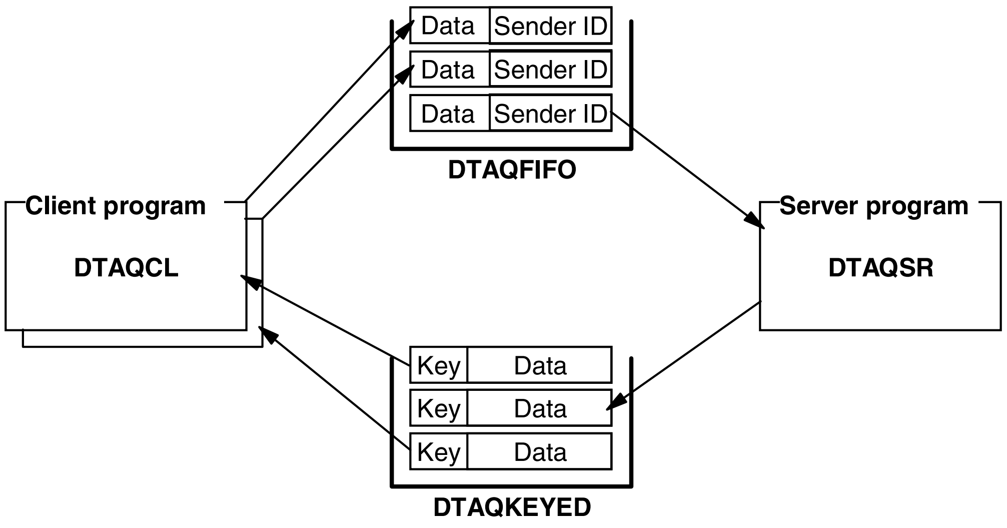


Figure 22. Communication between programs using data queues

Program DTAQCL is a client that requires some support from the server program DTAQSR. They communicate through two data queues: DTAQFIFO and DTAQKEYED.

The server program DTAQSR should be submitted as a batch job. It waits as long as the first message arrives into data queue DTAQFIFO. After receiving this message, the server sends response to the client into data queue DTAQKEYED. For testing purposes, in our example, the server receives only one message and ends. In a real application, the server would wait in an endless loop for next message to serve.

Data queue DTAQFIFO supports FIFO sequence and provides sender information. It is defined with the following command:

```
CRTDTAQ DTAQ(DTAQFIFO) MAXLEN(40) SENDERID(*YES)
```

Data queue DTAQKEYED supports a keyed sequence without sender information and is defined with the following command:

```
CRTDTAQ DTAQ(DTAQKEYED) MAXLEN(40) SEQ(*KEYED) KEYLEN(6) SENDERID(*NO)
```

Keyed access performance

The sequence algorithm is not efficient. It is an On**2 (OrderN squared) sort. The more keyed entries that exist in the data queue, the slower it is.

The sort or sequencing of entries happens at different times. If the system is IMPI, the sort happens when entries are enqueued. Adding entries gets slower as more keyed entries are added. If the system is RISC, the sort occurs when entries are dequeued. This results in a quite an exceptional delay to retrieve the first entry if many (as in thousands) entries are on the data queue.

It is also worth mentioning that queues only grow in size. Although entries are removed from the queue, the space is not reclaimed. If an application adds entries faster than they can be processed, the queue will grow in size and should be deleted and re-created to recover DASD space.

5.2.3.2 Source code for program DTAQCL

The client program DTAQCL sends a message to the data queue DTAQFIFO, which provides sender information, and waits for an answer on the data queue DTAQKEYED. To access a corresponding answer, the program uses its own job number, which is used as a key for data queue DTAQKEYED.

```
 ***********************************************************************
 *  Filename DTAQCL from APISRC in RPGISCOOL
 *  RPG program DTAQCL, sends message to FIFO data queue,
 *                      receives message from KEYED data queue
 ***********************************************************************
 *
 * Include prototypes                                                  1
D/Copy RPGISCOOL/APISRC,DTAQPROTO
 *
 * Program variable definitions
D  Length         S              5P 0
D  DataRcv        S             40A
D  Sender         S             44A
 *
D  DataSnd        C                   'Hello Server, how are you today?'
D  WaitTime       C                   5
 *
 * Program status data structure to get job number                       2
D  ProgStat      SDS
D  JobNum               264    269A
 *-----------------------------------------------------------------
 *
 * Sends message to server                                       3
C                   CallP     SndDtaQ('DTAQFIFO' : 'RPGISCOOL'
C                                   : %Len(%Trim(DataSnd)) : DataSnd)
 *
 * Receives answer from server using JobNum as the key                 4
C                   CallP     RcvDtaQ('DTAQKEYED' : 'RPGISCOOL'
C                                   : Length : DataRcv : WaitTime
C                                   : 'EQ' : %Size(JobNum) : JobNum
C                                   : *zero : Sender )
 *
 * Displays received data
C                   If        DataRcv <> *blanks
C     DataRcv       Dsply
C                   EndIf
 *
C                   Eval      *InLR = *On
```

DTAQCL program notes

1 Copy prototype definitions from member DTAQPROTO. The prototype member is defined in 5.2.2.6, "RPG IV prototype for data queue APIs" on page 148.

2 The job number is extracted from the program status data structure.

3 Calling QSNDDTAQ API. The parameters expected by the API are defined in the prototype copy member. On this call, the optional parameters related to keyed Dtaq are not used (feature of the Options(*Nopass) keyword on the prototype definition).

4 Calling QRCVDTAQ API. The parameters expected by the API are defined in the prototype copy member. The Wait parameter specifies 5 seconds to wait for an answer. The key order is equal, and the job number is used as a key. The sender length is initialized to zero because the sender information is not used.

5.2.3.3 Source code for program DTAQSR

The server program DTAQSR receives this message from the data queue DTAQFIFO, extracts the job number from sender information, and sends the response back to the data queue DTAQKEYED with a key equal to the job number of client program.

```
 **********************************************************************
 *  Filename DTAQSR from APISRC in RPGISCOOL
 *  RPG program DTAQSR, receives message from FIFO data queue,
 *                      sends message to KEYED data queue
 **********************************************************************
 *
 *-------------------------------------------------------------------
 * 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.
 *
 * Include prototypes                                                1
D/Copy RPGISCOOL/APISRC,DTAQPROTO
 *
 * Program variable definitions
D  Length         S              5P 0
D  Data           S             40A
D  KeyBuf         S              6A
D  Sender         S             44A
 *
D  WaitTime       C                   -1
 *-------------------------------------------------------------------
 *
 * Receives message from client                        2
C                   CallP     RcvDtaQ('DTAQFIFO' : 'RPGISCOOL'
C                                     : Length : Data : WaitTime
C                                     : *blank : *zero : KeyBuf
C                                     : %Size(Sender) : Sender)
 *
 * Sends answer to client                                3
C                   Eval      Data = 'Hello Client '
C                                    + %Subst(Sender : 29 : 6)
C                                    + ', thanks for calling'
 *
C                   CallP     SndDtaQ('DTAQKEYED' : 'RPGISCOOL'
C                                     : %Len(%Trim(Data)) : Data
C                                     : 6 : %Subst(Sender : 29 : 6))
 *
C                   Eval      *InLR = *On
```

DTAQSR program notes

1 Copy prototype definitions from member DTAQPROTO. The prototype member is defined in 5.2.2.6, “RPG IV prototype for data queue APIs” on page 148.

2 Calling QRCVDTAQ API. The parameters expected by the API are defined in the prototype copy member. The Wait parameter specifies a negative value, which means unlimited wait. The key length is initialized to 0 because key is not used. The sender information is required, and the sender length must be greater than 8.

3 Preparing data and calling QSNDDTAQ API. The parameters expected by the API are defined in the prototype copy member. The job number is extracted from the sender information and is used as a key.

Try it yourself

You can try this example by compiling the code from this section on your AS/400 system. Use the following command to create the programs:

```
CRTBNDRPG PGM(RPGISCOOL/DTAQCL) SRCFILE(RPGISCOOL/APISRC)
CRTBNDRPG PGM(RPGISCOOL/DTAQSR) SRCFILE(RPGISCOOL/APISRC)
```

Create two data queues using the following commands:

```
CRTDTAQ DTAQ(RPGISCOOL/DTAQFIFO) MAXLEN(40) SENDERID(*YES)
CRTDTAQ DTAQ(RPGISCOOL/DTAQKEYED) MAXLEN(40) SEQ(*KEYED) KEYLEN(6)
SENDERID(*NO)
```

To run the programs, use the following commands:

```
SBMJOB CMD(CALL PGM(RPGISCOOL/DTAQSR))
CALL PGM(RPGISCOOL/DTAQCL)
```

The following answer appears on your screen:

```
Hello Client 075608, thanks for calling
```

5.3 User space APIs

User spaces are objects that can be used to contain a stream of user-defined data. They are permanent objects that are located in either the system domain or the user domain. They have an object type of *USRSPC and a maximum size of 16 MB. You can save and restore user spaces to other systems. However, if the user spaces contain pointers, you cannot restore the pointers even if you want to restore them to the same system.

System value QALWUSRDMN specifies the libraries that can contain user-domain user spaces, either *ALL to allow them in all libraries or a list of up to 50 libraries, which must include QTEMP. This system value is only important when running level 40 or 50 security.

If the allow user domain (QALWUSRDMN) system value contains only the QTEMP library, you can only use the user space through APIs unless the user space is in QTEMP.

User or system domain?

All objects are assigned a domain attribute when they are created. A domain is a characteristic of an object that controls how programs can access the object. Once set, the domain remains in effect for the life of the object. For more information about the differences between the user and system domains, see *AS/400 Advanced Series System API Programming*, SC41-5800.

User spaces can be used for the following purposes:

- Store large amounts of data. You can create a user space as large as 16 MB, while data areas are limited to 2000 bytes.
- Save information in user space object, and save and restore the object with the information in it using CL commands.

- Pass data from job to job or from system to system.
- List APIs require user spaces to generate their output data.
- Store pointers.

5.3.1 List of user space APIs

The access to user spaces is possible through the group of APIs, which can be used from any high-level language program, including RPG IV. The following sections detail the user space APIs and how to use them.

5.3.1.1 Create User Space (QUSCRTUS) API

The Create User Space (QUSCRTUS) API creates a user space in either the user domain or the system domain. A system-domain user space cannot be saved to a release prior to Version 2 Release 3 Modification 0. A user-domain user space can be directly manipulated with machine interface (MI) instructions or can be accessed by using system APIs.

On systems with a QSECURITY system value of 40 or greater, applications can only access system-domain user spaces using APIs.

The user space objects that you create are larger than or equal to the size specified. They have a fixed length and can be extended or truncated using the Change User Space Attributes (QUSCUSAT) API.

For performance reasons, the *USRSPC object is created before checking to see if it already exists in the specified library. If you have an application using this API repeatedly, even if you are using *NO for the replace parameter, permanent system addresses are used.

This API has required and optional parameters. Table 22 shows the required parameters.

Table 22. Required parameter group for QUSCRTUS

Number	Description	Use	Data type
1	Qualified user space name	Input	Char(20)
2	Extended attribute	Input	Char(10)
3	Initial size	Input	Integer(10)
4	Initial value	Input	Char(1)
5	Public authority	Input	Char(10)
6	Text description	Input	Char(50)

Qualified user space name contains the user space name in the first 10 characters, and the name of the library where the user space is located in the second 10 characters. The only special value supported for the library name is *CURLIB.

User spaces created in the QTEMP and QRPLOBJ libraries are not forced to permanent storage. They are deleted when those libraries are cleared at sign-off and system IPL, respectively.

The extended attribute must be a valid *NAME. You can enter this parameter in uppercase, lowercase, or mixed case. The API converts it to uppercase.

The initial size of the user space being created is the value between 1 byte to 16,776,704 bytes.

You will achieve the best performance if you set the initial value byte to X'00'.

Public authority is the authority given to users who do not have specific private or group authority to the user space. Once the user space has been created, its public authority stays the same when it is moved to another library or restored from backup media.

Table 23. Optional parameter group 1 for QUSCRTUS

Number	Description	Use	Data type
7	Replace	Input	Char(10)
8	Error code	I/O	Char(*)

An existing user space can be replaced. Valid values are *YES or *NO. If the user space already exists, it is replaced by a new user space of the same name and library, and is subject to the same authorities.

Error code defines the structure in which to return error information. For the format of the structure, see “Error code parameter” on page 143.

Table 24. Optional parameter group 2 for QUSCRTUS

Number	Description	Use	Data type
9	Domain	Input	Char(10)

The user space can be created either in the system or user domain. If this parameter is not specified, the value of *DEFAULT is assumed by the API. Valid values for this parameter are:

***DEFAULT** Allows the system to decide into which domain the object should be created.

***SYSTEM** Creates the user space object into the system domain. The API can always create a user space into the system domain regardless of the security level in effect. However, you must use APIs to access system-domain user spaces if you are running at security level 40 or greater.

***USER** Attempts to create the user space object into the user domain. This is not always possible. If the library into which you are creating the user space does not appear in the QALWUSRDMN system value, the API cannot create the user space into the user domain. An error is returned.

5.3.1.2 Delete User Space (QUSDLTUS) API

The Delete User Space (QUSDLTUS) API deletes user spaces created with the Create User Space (QUSCRTUS) API. The QUSDLTUS API performs the same function as the Delete User Space (DLTUSRSPC) command.

All parameters for this API are required. See Table 25.

Table 25. Required parameter group for QUSDLTUS

Number	Description	Use	Data type
1	Qualified user space name	Input	Char(20)
2	Error code	I/O	Char(*)

The Qualified user space name parameter is described in Table 22 on page 154. The Error code parameter is described in “Error code parameter” on page 143.

5.3.1.3 Retrieve Pointer to User Space (QUSPTRUS) API

The Retrieve Pointer to User Space (QUSPTRUS) API retrieves a pointer to the contents of a user-domain user space. The data in that user space then can be directly manipulated by high-level language programs that support pointers, such as RPG IV. The QUSPTRUS API will not return a pointer to a system-domain user space. You must use system APIs to access system-domain user spaces.

This API has required and optional parameters, which are shown in Table 26.

Table 26. Required parameter group for QUSPTRUS

Number	Description	Use	Data type
1	Qualified user space name	Input	Char(20)
2	Return pointer	Output	Pointer

QUSPTRUS API returns the pointer to the user space in the variable defined as the pointer.

Table 27. Optional parameter for QUSPTRUS

Number	Description	Use	Data type
3	Error code	I/O	Char(*)

5.3.1.4 Retrieve User Space (QUSRTVUS) API

The Retrieve User Space (QUSRTVUS) API allows you to retrieve the contents of a user space. The QUSRTVUS API does not retrieve descriptive information about the user space object, such as its size. To retrieve information about the attributes of a user space, use QUSRUSAT API.

This API has required and optional parameters, which are shown in Table 28.

Table 28. Required parameter group for QUSRTVUS

Number	Description	Use	Data type
1	Qualified user space name	Input	Char(20)
2	Starting position	Input	Integer(10)
3	Length of data	Input	Integer(10)
4	Receiver variable	Output	Char(*)

Data from the user space at the starting position, in the length defined by the Length parameter, is moved to the receiver variable.

Table 29. Optional parameter for QUSRTVUS

Number	Description	Use	Data type
5	Error code	I/O	Char(*)

5.3.1.5 Change User Space (QUSCHGUS) API

The Change User Space (QUSCHGUS) API changes the contents of the user space (*USRSPC) object by moving a specified amount of data to the object. This API allows you to change the contents of a user space if you are using either:

- A language that does not support pointers
- System-domain user spaces

This API has required and optional parameters, which are shown in Table 30.

Table 30. Required parameter group for QUSCHGUS

Number	Description	Use	Data type
1	Qualified user space name	Input	Char(20)
2	Starting position	Input	Integer(10)
3	Length of data	Input	Integer(10)
4	Input data	Input	Char(*)
5	Force changes to auxiliary storage	Input	Char(1)

The input data in the length defined by the Length parameter are placed into the user space from the starting position.

The valid values for forcing changes to auxiliary storage are:

0	Does not force changes.
1	Forces changes asynchronously.
2	Forces changes synchronously.

Table 31. Optional parameter for QUSCHGUS

Number	Description	Use	Data type
6	Error code	I/O	Char(*)

Error code is explained in “Error code parameter” on page 143.

5.3.1.6 Retrieve User Space Attributes (QUSRUSAT) API

The Retrieve User Space Attributes (QUSRUSAT) API retrieves information about the current attributes and the current operational statistics of the user space.

All parameters for this API are required. They are shown in Table 32 on page 158.

Table 32. Required parameter group for QUSRUSAT

Number	Description	Use	Data type
1	Receiver variable	Output	Char(*)
2	Length of receiver variable	Input	Integer(10)
3	Format name	Input	Char(8)
4	Qualified user space name	Input	Char(20)
5	Error code	I/O	Char(*)

The receiver variable receives the requested information in the format defined by format name parameter. The valid format name for this API is SPCA0100.

SPCA0100 format

Table 33 shows the information about a user space returned for the format SPCA0100.

Table 33. SPCA0100 format

From	To	Description	Type
1	4	Bytes returned	Integer(10)
5	8	Bytes available	Integer(10)
9	12	Space size	Integer(10)
13	13	Automatic extendibility	Char(1)
14	14	Initial value	Char(1)
15	24	User space library name	Char(10)

The size of the user space object is returned in bytes.

Automatic extendibility specifies whether the space is extended automatically by the system when the end of the space is encountered:

0	Space is not automatically extendible.
1	Space is automatically extendible.

The initial value defines the character to which future extensions of the user space will be set.

5.3.1.7 Change User Space Attributes (QUSCUSAT) API

The Change User Space Attributes (QUSCUSAT) API changes the attributes of a user space object. This API can be used to:

- Extend or truncate a user space
- Mark or unmark the user space as automatically extendible by the system
- Change the initial value to which future extensions of the user space will be set

All parameters for this API are required. They are shown in Table 34.

Table 34. Required parameter group for QUSCUSAT

Number	Description	Use	Data type
1	Returned library name	Output	Char(10)
2	Qualified user space name	Input	Char(20)
3	Attributes to change	Input	Char(*)
4	Error code	I/O	Char(*)

If the user space attributes are successfully changed, the name of the library in which the user space was found is returned in the first parameter.

The attributes of the user space object that you want to change are written in a special variable format. For more information, refer to the *System API Reference*, SC41-5865.

5.3.2 Programming directly with user space APIs

The following code snippet illustrates the implementation of the two most often used user space APIs: Create User Space (QUSCRTUS) and Retrieve Pointer to User Space (QUSPTRUS):

```
 * Prototype for API QUSCRTUS - Create user space
D CrtUsrSpc       PR                  EXTPGM('QUSCRTUS')
D  QualUserSpc                  20A
D  ExtAttr                      10A
D  InitSize                     10I 0
D  InitValue                     1A
D  PubAuth                      10A
D  Text                         50A
D  Replace                      10A
D  Error                       256A
D  Domain                       10A
 *
 * Prototype for API QUSPTRUS - Retrieve pointer
D RtvPtr          PR                  EXTPGM('QUSPTRUS')
D  QualUserSpc                  20A
D  Pointer                        *
D  Error                       256A
 *
 * Program variable definitions
D  QualUserSpc    S             20A   Inz('PARTS     *CURLIB   ')
D  ExtAttr        S             10A   Inz(*BLANKS)
D  InitSize       S             10I 0 Inz(8192)
D  InitValue      S              1A   Inz(*BLANK)
D  PubAuth        S             10A   Inz('*USE')
D  Text           S             50A   Inz('User space example')
D  Replace        S             10A   Inz('*YES')
D  Error          S            256A
D  Domain         S             10A   Inz('*USER')
D  Pointer        S               *
 *-------------------------------------------------------------------
 *
 * Create user space calling QUSCRTUS
C                   CallP     CrtUsrSpc(QualUserSpc:ExtAttr:InitSize:
C                             InitValue:PubAuth:Text:Replace:Error:
C                             Domain)
 *
 * Retrieve pointer calling QUSPTRUS
C                   CallP     RtvPtr(QualUserSpc:Pointer:Error)
```

First, we define the prototypes for both APIs, so that we can call them using the CALLP operation. We also define program variables and their initial values.

Then we invoke the API Create User Space (QUSCRTUS) to create the user space named PARTS in current library. The size of this space is 8192 bytes, and it is initially filled with blanks. Public authority for this object is *USE. If such an object already exists, it is replaced.

Finally, we call API Retrieve Pointer to User Space (QUSPTRUS), which returns the pointer to the created user space. Using this pointer, we can access the user space and either store or retrieve data from it.

5.3.3 Simplifying user space APIs programming with wrappers

In the previous example, we saw that user space APIs can have a lot of parameters, especially the QUSCRTUS API used to create the user space. If these APIs are used often, the programmer can simplify the API use. They can create their own procedures to mask the complexity of system API parameters. This is called *wrappering*.

The benefits of using these procedures rather than coding to the APIs directly are:

- Only one person has to understand the complexities of the APIs. Everyone else in the shop can use a simpler interface. They do not have to worry about the mechanics of the process.
- Parameters can be re-sequenced to more closely match the way in which programmer would use the APIs.
- Provide standardized error handling routines.

These procedures are usually placed in a separate module, which must be bound with the module that uses them. It is also possible to include these procedures into a service program, which is then bound to other modules.

5.3.3.1 Source code for prototypes SPACEPROTO

Source member SPACEPROTO contains prototypes for calling the user space APIs QUSCRTUS and QUSPTRUS, and prototypes for calling the wrapper procedures CreateSpace, GetSpacePtr, and CheckPart:

```
 *  Filename SPACEPROTO from APISRC in RPGISCOOL
 *  Prototype for QUSCRTUS API
D QUSCrtUS        Pr                  Extpgm('QUSCRTUS')                1
D  QualObjName                  20A   Const
D  ExtAttr                      10A   Const
D  InitSize                     10U 0 Const
D  InitVal                       1A   Const
D  Authority                    10A   Const
D  Text                         50A   Const
D  Replace                      10A   Const
D  ErrorCode                   272A
D  Domain                       10A   Const

 *  Prototype for QUSPTRUS API                                          2
D QUSPtrUS        Pr                  Extpgm('QUSPTRUS')
D  QualObjName                  20A   Const
D  SpacePtr                       *
D  ErrorData                   272A

 *  Prototype for CreateSpace procedure
 *  First parameter is the space name, all others are optional
D CreateSpace     Pr             N                                      3
D  SpaceName                    10A   Value
D  LibraryName                  10A   Value Options(*NoPass)
D  InitSize                     10U 0 Value Options(*NoPass)
D  Text                         50    Value Options(*NoPass)
D  Replace                      10    Value Options(*NoPass)
D  ExtAttr                      10    Value Options(*NoPass)
D  InitVal                       1    Value Options(*NoPass)
D  Authority                    10    Value Options(*NoPass)
```

```
D  Domain                      10    Value Options(*NoPass)

 *  Prototype for GetSpacePtr procedure
 *  First parameter is space name, second (optional) is library name
D GetSpacePtr     Pr             *                                    4
D  SpaceName                   10A   Const
D  LibraryName                 10A   Const Options(*NoPass)

 * Prototype for CheckPart procedure used in Parts search
D CheckPart       Pr           10I 0                                 5
D  SearchKey@                    *   Value
D  Candidate@                    *   Value
```

SPACEPROTO program notes

1 Prototype for calling API QUSCRTUS with nine parameters.

2 Prototype for calling API QUSPTRUS with three parameters.

3 Prototype for the procedure CreateSpace, which returns an indicator variable to inform the caller about the success of the requested function. Only the first parameter SpaceName is mandatory. All others are optional and provided by the procedure. All parameters following the first optional parameter must also have `OPTIONS(*NOPASS)` specified.

4 Prototype for procedure GetSpacePtr, which returns a pointer variable. It has two parameters: SpaceName and LibraryName. The first parameter is mandatory, and the second, defined with keyword Options(*NoPass), is optional. The parameter defined with CONST cannot be modified by the called program or procedure.

5 Prototype for CheckPart procedure defined inside the program SHOWPARTS.

5.3.3.2 Example of masking procedures with SPACEPROCS

This section provides the source code for the program SPACEPROCS, which illustrates how these masking procedures can be created. Our example shows a module with two procedures for calling the two most used user space APIs.

The procedure CreateSpace acts as a *wrapper* for the system API Create User Space (QUSCRTUS). The procedure GetSpacePtr acts as a *wrapper* for the system API Retrieve Pointer to User Space (QUSPTRUS).

This module must be compiled (PDM option 15) and bound later with application programs that use the procedures:

```
 *  Filename SPACEPROCS from APISRC in RPGISCOOL
H NoMain                                                             1

 * Include prototypes
D/Copy RPGISCOOL/APISRC,SPACEPROTO                                    2

 * Following fields may be used in multiple procedures
D Authority       S             10
D Domain          S             10
D ErrorMsg        S             36A   Inz
D ExtAttr         S             10
D InitSize        S             10U 0
D InitVal         S              1
D LibraryName     S             10A
D Replace         S             10
D QualObjName     S             20A
D SpaceName       S             10A
D Text            S             50
D Wait            S              1A

 * This DS is used by the QUS... APIs to return error information
 * It is Exported to allow user procs to handle errors if they wish
D  ErrorInfo      DS                  Export                         3
D   BytesAvail                 10U 0 Inz(%Size(ErrorInfo))
```

```
D   BytesUsed                   10U 0
D   ExceptionId                  7A
D                                1A
D   ExceptionData...
D                              256A

 * Default values for parameters that can be omitted
D DftLibName      C                   '*CURLIB'                       4
D DftInitSize     C                   32000
D DftExtAttr      C                   *Blanks
D DftInitVal      C                   X'00'
D DftAuthority    C                   '*USE'
D DftText         C                   *Blanks
D DftReplace      C                   '*NO'
D DftDomain       C                   '*DEFAULT'

 ****************** Start of procedures

 * Procedure GetSpacePtr - returns pointer to an existing user space
 *                        returns null pointer on error (not found)
P GetSpacePtr     B                   Export                          5
D GetSpacePtr     PI              *
D  SpaceName                    10A   Const
D  LibraryName                  10A   Const Options(*NoPass)

D UserSpacePtr    S               *

 * Use default value for Library name if not supplied
C                   If        %Parms < 2                              7
C                   Eval      QualObjName = SpaceName + DftLibName
C                   Else
C                   Eval      QualObjName = SpaceName + LibraryName
C                   EndIf

C                   CallP     QUSPtrUS(QualObjName:UserSpacePtr:
C                                      ErrorInfo)

 * Check for error and report
C                   If        BytesUsed > 0                           8
C                   Eval      ErrorMsg = 'QUSPTRUS returned error ' +
C                                      ExceptionId
C     ErrorMsg      Dsply                   Wait
C                   Return    *Null
C                   Else
C                   Return    UserSpacePtr
C                   EndIf
P GetSpacePtr     E

  * Procedure CreateSpace - returns an indicator which is on if the
 *                 returns null pointer on error (not found)
P CreateSpace     B                   Export                          6
D                 PI              N
D  PSpaceName                   10A   Value
D  PLibraryName                 10A   Value Options(*NoPass)
D  PInitSize                    10U 0 Value Options(*NoPass)
D  PText                        50    Value Options(*NoPass)
D  PReplace                     10    Value Options(*NoPass)
D  PExtAttr                     10    Value Options(*NoPass)
D  PInitVal                      1    Value Options(*NoPass)
D  PAuthority                   10    Value Options(*NoPass)
D  PDomain                      10    Value Options(*NoPass)

 * Use default values for any optional parameters that were not passed
C                   If        %Parms < 2                              7
C                   Eval      QualObjName = PSpaceName + DftLibName
C                   Else
C                   Eval      QualOBjName = PSpaceName + PLibraryName
C                   EndIf

C                   If        %Parms < 3
C                   Eval      InitSize = DftInitSize
C                   Else
C                   Eval      InitSize = PInitSize
C                   EndIf

C                   If        %Parms < 4
C                   Eval      Text = DftText
C                   Else
```

```
C                   Eval      Text = PText
C                   EndIf

C                   If        %Parms < 5
C                   Eval      Replace = DftReplace
C                   Else
C                   Eval      Replace = PReplace
C                   EndIf

C                   If        %Parms < 6
C                   Eval      ExtAttr = DftExtAttr
C                   Else
C                   Eval      ExtAttr = PExtAttr
C                   EndIf

C                   If        %Parms < 7
C                   Eval      InitVal = DftInitVal
C                   Else
C                   Eval      InitVal = PInitVal
C                   EndIf

C                   If        %Parms < 8
C                   Eval      Authority = DftAuthority
C                   Else
C                   Eval      Authority = PAuthority
C                   EndIf

C                   If        %Parms < 9
C                   Eval      Domain = DftDomain
C                   Else
C                   Eval      Domain = PDomain
C                   EndIf

C                   CallP     QUSCRTUS(QualObjName:ExtAttr:InitSize:
C                                      InitVal:Authority:Text:
C                                      Replace:ErrorInfo:Domain)

 * Check for error and report
C                   If        BytesUsed > 0                         8
C                   Eval      ErrorMsg = 'QUSCRTUS returned error ' +
C                                        ExceptionId
C     ErrorMsg      Dsply                   Wait
C                   Return    *Off
C                   Else
C                   Return    *On
C                   EndIf
P CreateSpace     E
```

SPACEPROCS program notes

1 Keyword NOMAIN on H specifications specifies that the cycle code is not generated for this module. In our case, we recommend this because it reduces the size of the program.

2 Copy prototype definitions from member SPACEPROTO.

3 This data structure is used by user space APIs to return error information. The keyword EXPORT allows user procedures to handle errors if they want.

4 These constants contain default values for optional parameters.

5 The beginning and procedure interface for the procedure GetSpacePtr. The keyword EXPORT allows the use of the procedure outside of this module.

6 The beginning and procedure interface for the procedure CreateSpace. The keyword EXPORT allows the use of the procedure outside of this module.

7 If optional parameters are not defined, provide default values and call the API.

8 Check for an error and display error message. Otherwise, return the requested value.

5.3.4 A user space example

Once procedures are prepared, we can use them in application programs. In our example, we use three simple programs designed to demonstrate the use of user spaces as an alternative to a database for table-type lookups. We do not have any performance data. A reduced number of I/O operations alone should be of significant benefit, particularly during batch operations which tend to be I/O constrained. Figure 23 shows the logic of our small application.

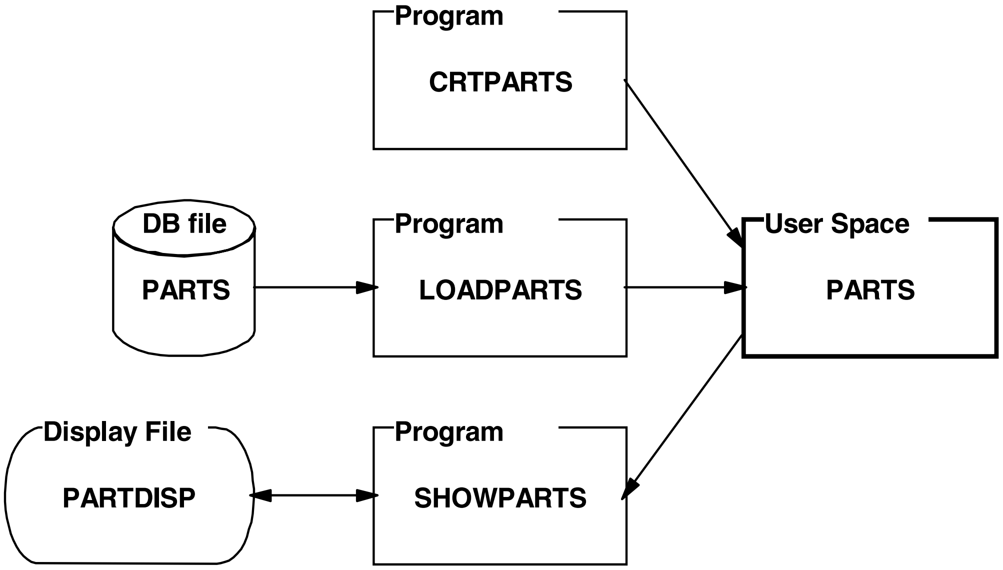


Figure 23. Programs used in the user space example

Note these points in regard to Figure 23:

- The program CRTPARTS creates a user space PARTS in the current library. If this object already exists, the program returns an error message.
- The program LOADPARTS reads records from database file PARTS and loads them into previously created user space PARTS.
- The program SHOWPARTS uses the display file PARTDISP as an interface with the end user. It looks for a requested part in the user space PARTS, and if found, it displays data. Otherwise, it sends an error message.

5.3.4.1 Source code for the program CRTPARTS

This program is used only once to create the user space. Once created, the user space remains in the library as a permanent object. It can be deleted any time by using the CL command DLTUSRSPC, or eventually by using a similar program that calls API QUSDLTUS (delete user space).

The source program must be compiled into a module, and then bound with the module SPACEPROCS to create an executable program. To achieve this, use the following commands:

- `CRTRPGMOD MODULE(CRTPARTS)`
- `CRTPGM PGM(CRTPARTS) MODULE(CRTPARTS SPACEPROCS)`

The source code for the program CRTPARTS is shown here:

```
 *  Filename CRTPARTS from APISRC in RPGISCOOL

 * Include prototypes
D/Copy RPGISCOOL/APISRC,SPACEPROTO                                  1
 *
D SpaceName       S             10A   Inz('PARTS')
D ErrorMsg        C                   'User Space PARTS already exists'

C                   If        Not CreateSpace(SpaceName)            2
C     ErrorMsg      Dsply
C                   EndIf

C                   Eval      *InLr = *ON
```

CRTPARTS program notes

1 Copy prototype definitions from member SPACEPROTO.

2 Procedure CreateSpace returns an indicator as a return value. If the function is successfully completed, the returned indicator is *On. We can call the procedure and check the returned value with the IF operation.

5.3.4.2 Source code for the program LOADPARTS

This program can be called to load all records from the database file PARTS into the user space. Each time the file content is changed. Once loaded, the user space can be used by any application program instead of opening and reading the PARTS file. The result should be better performance because access to the user space is faster than access to the database files.

The source program must be compiled into a module, and then bound with the module SPACEPROCS to create an executable program. To achieve this, use the following commands:

- `CRTRPGMOD MODULE(LOADPARTS)`
- `CRTPGM PGM(LOADPARTS) MODULE(LOADPARTS SPACEPROCS)`

The source code for program LOADPARTS follows:

```
 *  Filename LOADPARTS from APISRC in RPGISCOOL
FParts     IF   E           K Disk

 * Include prototypes
D/Copy RPGISCOOL/APISRC,SPACEPROTO                                  1

D PartRecord    E DS                  ExtName(Parts)

D PartEntry       S                   Based(PartPtr) Like(PartRecord)  2

D Library         S             10A   Inz('*CURLIB')
D SpaceName       S             10A   Inz('PARTS')

 * BasePtr will hold the base address of the User Space PARTS
 * At the beginning of the space is a count (Count) of the entries
D BasePtr         S               *                                 3
D Count           S              5P 0 Based(BasePtr)
D CountMessage    S             30A
D SpaceNotFound   C                   'User Space PARTS not found'

C                   Eval      BasePtr=GetSpacePtr(SpaceName:Library)   4

C                   If        BasePtr <> *Null                      5

C                   Eval      PartPtr = BasePtr + %Size(Count)
C                   Eval      Count = 0

C                   Dow       Not %Eof(Parts)

C                   Read      Parts                                 6
C                   If        Not %Eof(Parts)
```

```
C                   Eval      Count = Count + 1
C                   Eval      PartEntry = PartRecord
C                   Eval      PartPtr = PartPtr + %Size(PartRecord)
C                   EndIf

C                   EndDo

C                   Eval      CountMessage = %EditC(Count:'Z')          7
C                                           + ' Records loaded'
C     CountMessage  Dsply

C                   Else

 * Space was not found - inform user
C     SpaceNotFound Dsply
C                   EndIf

C                   Eval      *InLr = *ON
```

LOADPARTS program notes

1 Copy prototype definitions from the member SPACEPROTO.

2 The PartEntry variable is based on pointer PartPtr and has the same definition as a PartRecord data structure. It is used to load records from the PARTS file into the user space.

3 Another pointer, BasePtr, is used to hold the base address of the user space PARTS. At the beginning of the space is a variable Count, based on pointer BasePtr, which contains the number of loaded entries.

4 Procedure GetSpacePtr loads the pointer BasePtr with the address of the requested user space.

5 If BasePtr contains valid value (not *Null), we load also the pointer PartPtr, which is used to load part records into the user space.

6 In the DoW loop, we read all records from the PARTS file and load them into the user space. The PartEntry field is based on the PartPtr pointer, which is increased each time for the length of one part record, and always points to the next free entry.

7 At the end, a count message is prepared to inform a user about the number of loaded records into user space.

Physical file PARTS

The physical file PARTS is required by the program LOADPARTS. Here is the definition of the file:

```
A***************************************************************
A* PARTS from DBSRC in 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
```

To recreate our sample test execution, you can load the following data in the file PARTS:

```
Part Number  Part Description           Part Quantity  Part Price  Shipment
   12345      Hammer                             123        29.99    1999-05-01
   23456      Saw                                234        20.99    1999-04-01
   34567      Hatchet                            345        19.99    1999-03-01
   45678      Rasp                               456         9.99    1999-02-01
```

5.3.4.3 Source code for the SHOWPARTS program

This program uses the user space PARTS loaded by the LOADPARTS program to perform the parts lookup. It gets a part number from the display file, looks for a part record in the user space, and if found, displays the parts data on the screen. To perform the actual lookup process, the program uses the C library routine bsearch(). You can see the prototype for bsearch() in “SHOWPARTS program notes” on page 168.

To create an executable program, the source program must be first compiled into a module. Then, it must be bound with the SPACEPROCS module and the QC2UTIL1 service program, which contains the C function bsearch. To achieve this, use the following commands:

- `CRTRPGMOD MODULE(SHOWPARTS)`
- `CRTPGM PGM(SHOWPARTS) MODULE(SHOWPARTS SPACEPROCS) BNDSRVPGM(QC2UTIL1)`

You can also use the binding directory QC2LE instead of the service program directly (BNDDIR(QC2LE)).

The source code for program SHOWPARTS follows:

```
 *  Filename SHOWPARTS from APISRC in RPGISCOOL
FPartDisp  CF   E             WorkStn IndDs(DispInd)

 * Include prototypes
D/Copy RPGISCOOL/APISRC,SPACEPROTO                                     1
D/Copy RPGISCOOL/APISRC,C_Protos                                       2

D DispInd         DS                                                   3
D  Exit                  3      3N
D  FoundMatch           50     50N

D PartRecord    E DS                  ExtName(Parts) Based(PartPtr)    4
D DummyPart     E DS                  ExtName(Parts) Inz Prefix(D_)

 * BasePtr will hold the base address of the User Space PARTS
 * At the beginning of the space is a count (Count) of the entries
D PartSpace       DS                  Based(BasePtr)                   5
D  Count                         5P 0
D  PartData                           Like(PartRecord) Dim(20)

D Library         S             10A   Inz('*CURLIB')
D SpaceName       S             10A   Inz('PARTS')
D True            C                   *On
D False           C                   *Off

C                   Eval      BasePtr=GetSpacePtr(SpaceName:Library)   6

 * If no space found error message is already sent, so set for Exit
C                   If        BasePtr = *Null
C                   Eval      Exit = True
C                   EndIf

 * This statement loads pointer required to display DispRec
C                   Eval      PartPtr = %Addr(DummyPart)               7

C                   DoU       Exit = True

C                   ExFmt     DispRec
C                   If        Exit
C                   Iter
C                   Endif

C                   Eval      PartPtr=FindIt(%Addr(I_PartNum):         8
C                                            %Addr(PartData):Count:
C                                            %Len(PartRecord):
C                                            %PAddr('CHECKPART'))

C                   If        PartPtr <> *Null
C                   Eval      FoundMatch = True
C                   Else
```

```
C                   Eval      FoundMatch = False
C                   Eval      PartPtr = %Addr(DummyPart)              9
C                   EndIf

C                   EndDo

C                   Eval      *InLR = True

 * Procedure CheckPart - used by search to determine key match
P CheckPart       B                   Export                          10
D CheckPart       PI            10I 0
D  SearchKey@                     *   Value
D  Candidate@                     *   Value

D SearchKey       S                   Based(SearchKey@) Like(I_PartNum)
D Candidate     E DS                  Based(Candidate@) ExtName(Parts)

 * The routine compares two elements passed and returns Low/High/Equal
C                   If        SearchKey < PartNum
C                   Return    Low
C                   Else
C                   If        SearchKey > PartNum
C                   Return    High
C                   Else
C                   Return    Equal
C                   EndIf
C                   EndIf
P                 E
```

SHOWPARTS program notes

1 Copy prototype definitions from the member SPACEPROTO.

2 In this program, we use the procedure FindIt, which invokes the C function bsearch. The Copy statement is needed to include the prototype for the procedure FindIt from the member C_PROTOS from APISRC in RPGISCOOL. As a result, the following code is copied:

```
D FindIt          PR              *   ExtProc('bsearch')
D  LookFor                        *   Value
D  DataStart                      *   Value
D  Elements                     10U 0 Value
D  Size                         10U 0 Value
D  CompFunc                       *   ProcPtr Value
```

3 Indicator data structure for the display file.

4 PartRecord is a data structure based on the pointer PartPtr, which contains the found record. DummyPart is a dummy data structure used only to initialize the PartPtr either before the first use or in the case when part is not found and procedure FindIt returns a null pointer.

5 PartSpace is a data structure based on pointer BasePtr, which maps the content of the user space. PartData is an array with 20 elements used in the search procedure FindIt.

6 Procedure GetSpacePtr loads the pointer BasePtr with the address of the user space PARTS.

7 PartPtr pointer must be initialized before the display file record can be displayed.

8 Procedure FindIt invokes the C function bsearch and transfers to it the address of the search argument field (I_PartNum), address of the PartData array containing all part records, the number of available records, the record size, and the address of the procedure CheckPart used for key comparison.

9 If the parts lookup was successful, pointer PartPtr is loaded with a valid value. Otherwise, we have to initialize it using a dummy data structure.

10 The CheckPart procedure is used by the search function FindIt to determine the key match. The procedure compares two elements passed as pointers and returns an integer value for low (-1), high (1), or equal (0). These integer values are defined in the copy member C_PROTOS.

Display file PARTDISP

Here is the definition of the display file PARTDISP, which is required when compiling program SHOWPARTS:

```
A                                       DSPSIZ(24 80 *DS3)
A                                       REF(RPGISCOOL/PARTS PARTR)
A                                       INDARA CF03(03)
A          R DISPREC
A                                  1  2DATE
A                                       EDTCDE(Y)
A                                  1 14TIME
A                                  1 61USER
A                                  1 73SYSNAME
A                                  4 26'Test Parts Table Lookup'
A                                  7 23'Part Number: . . .'
A            I_PARTNUM R        B  7 42REFFLD(PARTNUM)
A  50        PARTNUM   R        O  7 49REFFLD(PARTNUM)
A  50                              9 23'Part Name: . . . .'
A  50        PARTDES   R        O  9 42REFFLD(PARTDES)
A  50                             11 23'Part Quantity: . .'
A  50        PARTQTY   R        O 11 42REFFLD(PARTQTY)
A  50                             13 23'Part Price:  . . .'
A  50        PARTPRC   R        O 13 42REFFLD(PARTPRC)
A  50                             15 23'Shipping Date: . .'
A  50        PARTDAT   R        O 15 42REFFLD(PARTDAT)
 *
A N50                             21 23'No entry found for requested Part'
A                                       DSPATR(HI)
A                                       DSPATR(BL)
A  50                             22 23'Requested Part found              '
A                                       DSPATR(BL)
A                                       DSPATR(HI)
A                                 24  2'F3=Exit'
```

Try it yourself

You can try this example by compiling the code from this section on your AS/400 system. Use the following commands to create the modules. The physical file PARTS and display file PARTDISP must be created prior to creating the modules.

```
CRTPF FILE(RPGISCOOL/PARTS) SRCFILE(RPGISCOOL/DBSRC)
CRTDSPF FILE(RPGISCOOL/PARTDISP) SRCFILE(RPGISCOOL/APISRC)

CRTRPGMOD MODULE(RPGISCOOL/SPACEPROCS) SRCFILE(RPGISCOOL/APISRC)
CRTRPGMOD MODULE(RPGISCOOL/CRTPARTS) SRCFILE(RPGISCOOL/APISRC)
CRTRPGMOD MODULE(RPGISCOOL/LOADPARTS) SRCFILE(RPGISCOOL/APISRC)
CRTRPGMOD MODULE(RPGISCOOL/SHOWPARTS) SRCFILE(RPGISCOOL/APISRC)
```

Create the programs by using the following commands:

```
CRTPGM PGM(RPGISCOOL/CRTPARTS) MODULE(RPGISCOOL/CRTPARTS
RPGISCOOL/SPACEPROCS)
CRTPGM PGM(RPGISCOOL/LOADPARTS) MODULE(RPGISCOOL/LOADPARTS
RPGISCOOL/SPACEPROCS)
CRTPGM PGM(RPGISCOOL/SHOWPARTS) MODULE(RPGISCOOL/SHOWPARTS
RPGISCOOL/SPACEPROCS) BNDSRVPGM(QC2UTIL1)
```

Create the display file by using the following commands:

```
CRTDSPF FILE(RPGISCOOL/PARTDISP) SRCFILE(RPGISCOOL/APISRC) SRCMBR(PARTDISP)
```

To run the programs, use the following commands:

```
CALL PGM(RPGISCOOL/CRTPARTS)
CALL PGM(RPGISCOOL/LOADPARTS)
CALL PGM(RPGISCOOL/SHOWPARTS)
```

In the display file PARTDISP, you can enter a part number from the list of parts found in the physical file PARTS (which have been loaded in the user space). The content is described in “Physical file PARTS” on page 166. For example, you may enter:

```
Part Number: . . . 12345
```

Then, you are shown the part description, as loaded in the user space.

5.4 Message handling APIs

On the AS/400 system, communication between procedures or programs, between jobs, between users, and between users and procedures or programs occurs through messages. A message can be predefined or immediate:

Predefined message
Created and exists outside the program that uses it. They are stored in message files uniquely identified by a seven-character code and are defined by a message description.

Immediate message
Created by the sender at the time it is sent, and is not stored in a message file.

All messages on the system are sent to a message queue. The system user or program associated with the message queue receives the message from the queue. A message queue is automatically supplied for each display station and each user profile.

Job message queues are supplied for each job running on the system. Each job is given an external message queue (*EXT). Each call of an OPM program or ILE procedure within the job has its own call message queue. The external message queue (*EXT) is used to send messages between an interactive job and the workstation user.

In addition to these message queues, you can create your own user message queues for sending messages to system users and between application programs.

5.4.1 Message types

The OS/400 program divides messages into types according to their use. These message types are categories based on the message's purpose. The sender of the message determines its type when sending the message. The message handling APIs use these message types in defining and working with messages:

- **Completion (*COMP)**
  Reports the successful completion of a task.
- **Diagnostic (*DIAG)**
  Describes errors in processes or input data. When an error occurs, a program usually sends an escape message, which causes the task to end abnormally. One or more diagnostic messages can be sent before the escape message to describe the error.
- **Escape (*ESCAPE)**
  Indicates a condition causing a program to end abnormally, without completing its work.
- **Informational (*INFO)**
  Conveys information without asking for a reply.
- **Inquiry (*INQ)**
  Conveys information and asks for a reply.
- **Notify (*NOTIFY)**
  Describes a condition in the sending program requiring corrective action or a reply.
- **Reply (*RPY)**
  Responds to an inquiry or notify message.
- **Request (*RQS)**
  Requests a function from the receiving program.
- **Sender's copy (*COPY)**
  Copy of an inquiry or notify message, kept by the sender of the message.
- **Scope (*SCOPE)**
  Specifies a program to run when the program to which this message is sent completes.
- **Status (*STATUS)**
  Describes the status of work being done by a program.

5.4.2 List of message handling APIs

Message handling APIs let your applications work with AS/400 messages, and allow you to do the following tasks:

- Send messages to users or programs
- Receive messages from a message queue
- Handle errors
- Return message and message queue information
- Return message description and message file information

The message handling APIs include:

- **Change Exception Message (QMHCHGEM)**
  Changes an exception message (escape or notify) on a call message queue.
- **Control Job Log Output (QMHCTLJL)**
  Controls the production of a job log when the related job ends or when the job message queue becomes full and the print-wrap option is in effect for the job.
- **List Job Log Messages (QMHLJOBL)**
  Lists messages from the job message queue of a job, and returns it in a user space in the format specified in the parameter list, as seen in 5.4.2.3, “List Job Log Messages (QMHLJOBL) API” on page 177.
- **List Nonprogram Messages (QMHLSTM)**
  Lists messages from one or two nonprogram message queues, and returns it in a user space in the format specified in the parameter list.
- **Move Program Messages (QMHMOVPM)**
  Moves messages from one call message queue to the message queue of an earlier call stack entry in the call stack.
- **Open List of Job Log Messages (QGYOLJBL)**
  Lists messages from a job log, and returns them sorted by their sending date and time.
- **Open List of Messages (QGYOLMSG)**
  Provides information on messages for the current user, a specific user, or one specific nonprogram message queue.
- **Promote Message (QMHPRMM)**
  Promotes an escape or status message that was sent to a call stack entry.
- **Receive Nonprogram Message (QMHRCVM)**
  Receives a message from a nonprogram message queue, providing information about the sender of the message as well as the message itself. This function is similar to the Receive Message (RCVMSG) command with the MSGQ parameter.
- **Receive Program Message (QMHRCVPM)**
  Receives a message from a call message queue, and provides information about the sender of the message, as well as the message itself. See 5.4.2.2, “Receive Program Message (QMHRCVPM) API” on page 175. This function is similar to the Receive Message (RCVMSG) command with the PGMQ parameter.
- **Remove Nonprogram Messages (QMHRMVM)**
  Removes messages from nonprogram message queues. This function is similar to the Remove Message (RMVMSG) command with the MSGQ parameter.

- **Remove Program Messages (QMHRMVPM)**
  Removes messages from call message queues. This function is similar to the Remove Message (RMVMSG) command with the PGMQ parameter.
- **Resend Escape Message (QMHRSNEM)**
  Resends an escape message from one call message queue to the message queue of the previous call stack entry in the call stack.
- **Retrieve Message (QMHRTVM)**
  Retrieves the message text and other elements of a predefined message stored in a message file on your AS/400 system. This function is similar to the Retrieve Message (RTVMSG) command.
- **Retrieve Message File Attributes (QMHRMFAT)**
  Retrieves information about the attributes of a message file.
- **Retrieve Nonprogram Message Queue Attributes (QMHRMQAT)**
  Provides information about the attributes of a nonprogram message queue.
- **Retrieve Request Message (QMHRTVRQ)**
  Retrieves request messages from the current job's call message queue.
- **Send Break Message (QMHSNDBM)**
  Sends a message to a workstation for immediate display, interrupting the workstation user's task. This function is similar to the Send Break Message (SNDBRKMSG) command.
- **Send Nonprogram Message (QMHSNDM)**
  Sends a message to a system user or a message queue that is not associated with a specific program. This function is similar to the Send Program Message (SNDPGMMSG) command with the TOMSGQ parameter.
- **Send Program Message (QMHSNDPM)**
  Sends a message to the message queue of a call stack entry in the call stack as seen in 5.4.2.1, “Send Program Message (QMHSNDPM) API” on page 173. This function is similar to the Send Program Message (SNDPGMMSG) command with the TOPGMQ parameter.
- **Send Reply Message (QMHSNDRM)**
  Sends a response to an inquiry message. This function is similar to the Send Reply (SNDRPY) command.
- **Send Scope Message (QMHSNDSM)**
  Sends a scope message that allows a user to specify a program to run when your program or job is completed.

Our discussion is limited to only three of these APIs:

- Send Program Message (QMHSNDPM) API
- Receive Program Message (QMHRCVPM) API
- List Job Log Messages (QMHLJOBL) API

For additional information and description of other APIs, refer to *System API Reference: OS/400 Message Handling APIs*, SC41-5862.

5.4.2.1 Send Program Message (QMHSNDPM) API

The Send Program Message (QMHSNDPM) API sends a message to a call message queue or the external message queue. It allows the current call stack entry to send a message to its caller, a previous caller, or itself.

This API has required and optional parameters as shown in Table 35.

Table 35. Required parameter group for QMHSNDPM

Number	Description	Use	Data type
1	Message identifier	Input	Char(7)
2	Qualified message file name	Input	Char(20)
3	Message data or immediate text	Input	Char(*)
4	Length of message data or immediate text	Input	Integer(10)
5	Message type	Input	Char(10)
6	Call stack entry	Input	Char(*) or Pointer
7	Call stack counter	Input	Integer(10)
8	Message key	Output	Char(4)
9	Error code	I/O	Char(*)

Message identifier and Qualified message file name are used to identify the predefined message being sent. For an immediate message, these fields should be set to blanks.

The third parameter specifies the data to insert in the predefined message's substitution variables or the complete text of an immediate message.

For a message type, you can specify one of these values: *COMP, *DIAG, *ESCAPE, *INFO, *INQ, *NOTIFY, *RQS, or *STATUS.

Call stack entry together with Call stack count identify the program of the message queue to which the message is being sent.

Message key is assigned by the API and returned to the program.

Error code defines the structure in which to return error information. For the format of the structure, see “Error code parameter” on page 143.

Table 36. Optional parameter group 1 for QMHSNDPM

Number	Description	Use	Data type
10	Length of call stack entry	Input	Integer(10)
11	Call stack entry qualification	Input	Char(20)
12	Display program messages screen wait time	Input	Integer(10)

The Call stack entry qualification parameter is used when it is necessary to further identify the call stack entry.

Table 37. Optional parameter group 2 for QMHSNDPM

Number	Description	Use	Data type
13	Call stack entry data type	Input	Char(10)
14	Coded character set identifier	Input	Integer(10)

The Call stack entry data type parameter relates to the required parameter call stack entry and defines its type, either *CHAR or *PTR. If it is not specified, the assumed value is a character string.

5.4.2.2 Receive Program Message (QMHRCVPM) API

The Receive Program Message (QMHRCVPM) API receives a message from a call message queue or external message queue and returns information describing the message.

This API has required and optional parameters as shown in Table 38.

Table 38. Required parameter group for QMHRCVPM

Number	Description	Use	Data type
1	Message information	Output	Char(*)
2	Length of message information	Input	Integer(10)
3	Format name	Input	Char(8)
4	Call stack entry	Input	Char(*) or Pointer
5	Call stack counter	Input	Integer(10)
6	Message type	Input	Char(10)
7	Message key	Input	Char(4)
8	Wait time	Input	Integer(10)
9	Message action	Input	Char(10)
10	Error code	I/O	Char(*)

The first parameter defines the variable that receives the information returned, in the format specified in the Format name parameter, of the length specified in the Length of message information parameter.

For the format name, you can specify one of these names:

RCVM0100	Brief message information
RCVM0200	All message information
RCVM0300	All message information including sender information

Call stack entry together with Call stack count identify the program of the message queue to which the messages are to be received.

The Message type and Message key parameters are used together to specify the selection criteria for receiving messages.

Wait time specifies the amount of time to wait for the message to arrive in the queue so it can be received. The valid values are:

0	Do not wait for the message.
-1	Wait until the message arrives in the queue and is received, no matter how long it takes.
n	Wait n seconds for the message to arrive in the queue.

Message action defines the action to take after the message is received. The valid values are:

***OLD**	Keep the message in the message queue, and mark it as an old message.
***REMOVE**	Remove the message from the message queue.
***SAME**	Keep the message in the message queue without changing its new or old designation.

Table 39. Optional parameter group 1 for QMHRCVPM

Number	Description	Use	Data type
11	Length of call stack entry	Input	Integer(10)
12	Call stack entry qualification	Input	Char(20)

The Call stack entry qualification parameter is used when it is necessary to further identify the call stack entry.

Table 40. Optional parameter group 2 for QMHRCVPM

Number	Description	Use	Data type
13	Call stack entry data type	Input	Char(10)
14	Coded character set identifier	Input	Integer(10)

The Call stack entry data type parameter relates to the required parameter Call stack entry and defines its type as either *CHAR or *PTR. If it is not specified, the assumed value is a character string.

RCVM0100 format

The received message data is returned in the selected format. In our example, we use the simplest format RCVM0100. For other formats, refer to the IBM manual *System API Reference: OS/400 Message Handling APIs*, SC41-5862.

Table 41 shows the message information returned in the format RCVM0100.

Table 41. RCVM0100 format

From	To	Description	Type
1	4	Bytes returned	Integer(10)
5	8	Bytes available	Integer(10)
9	12	Message severity	Integer(10)
13	19	Message identifier	Char(7)
20	21	Message type	Char(2)
22	25	Message key	Char(4)
26	32	Reserved	Char(7)
33	36	CCSID conversion status indicator of message data or text	Integer(10)
37	40	CCSID of replacement data or impromptu message text	Integer(10)

From	To	Description	Type
41	44	Length of replacement data or impromptu message text returned	Integer(10)
45	48	Length of replacement data or impromptu message text available	Integer(10)
49	*	Replacement data or impromptu message text	Char(*)

Bytes returned defines the length of all information returned in the format. When you attempt to receive a message and the message is not found, the following results occur:

- The value of the bytes returned field is 8.
- The value of the bytes available field is 0.

The message severity, identifier, type, and key contain information about the received message.

The last field in this format contains the values for substitution variables in a predefined message, or the text of an impromptu message.

5.4.2.3 List Job Log Messages (QMHLJOBL) API

The List Job Log Messages (QMHLJOBL) API lists messages sent to the job message queue of a job. This API gets the requested message information and returns it in a user space in the format specified in the parameter list.

The generated list replaces any existing information in the user space. If the user space is not large enough to contain the data to be returned, the user space is increased to the maximum user space size allowed (16 MB) or the maximum amount of storage allowed to the user of the API.

This API has required parameters only, which are shown in Table 42.

Table 42. Required parameter group for QMHLJOBL

Number	Description	Use	Data type
1	Qualified user space name	Input	Char(20)
2	Format name	Input	Char(8)
3	Message selection information	Input	Char(*)
4	Size of message selection information	Input	Integer(10)
5	Format of message selection information	Input	Char(8)
6	Error code	I/O	Char(*)

The Qualified user space name parameter specifies the user space that receives the generated list, and the library in which it is located. The first 10 characters contain the user space name, and the second 10 characters contain the user space library.

The format of the returned message information is defined by the format name parameter. The valid format name is LJOB0100.

The message selection parameters determine the job message queue and messages to be selected. For the format name for the message selection information parameter, you can use JSLT0100 or JSLT0200. The formats are the same with the exception that the JSLT0200 format allows you to specify the CCSID for the returned message data.

JSLT0100 format

Table 43 shows the message selection information that can be defined using the format JSLT0100.

Table 43. JSLT0100 format

From	To	Description	Type
1	4	Maximum messages requested	Integer(10)
5	14	List direction	Char(10)
15	24	Qualified job name	Char(10)
25	34	Qualified user name	Char(10)
35	40	Qualified job number	Char(6)
41	56	Internal job identifier	Char(16)
57	60	Starting message key	Char(4)
61	64	Maximum message length	Integer(10)
65	68	Maximum message help length	Integer(10)
69	72	Offset to identifiers of fields to return	Integer(10)
73	76	Number of fields to return	Integer(10)
77	80	Offset to call message queue name	Integer(10)
81	84	Length of call message queue name	Integer(10)
The offsets to these fields are specified in the previous offset variables.		Identifiers of fields to return	Array(*) of Integer(10)
		Call message queue name	Char(*)

Maximum messages requested specifies the maximum number of messages to be returned. To list all messages in the job log in the specified list direction from the starting message key, use the special value of -1.

List direction specifies the direction to list messages. Valid values are *NEXT or *PRV.

To identify the job whose messages are to be listed, you can use either job name, user name and job number, or internal job identifier provided by the List Job (QUSLJOB) API. The value "*" in a job name is used to identify a current job.

Starting message key specifies a key to begin searching for messages. You can use these special values for the message key:

'00000000'X	Start searching from the oldest message in the queue.
'FFFFFFFF'X	Start searching from the newest message in the queue.

The maximum message length parameters specify the number of characters of text that this API returns. The special value of -1 defines that the maximum length will be used.

At the end of format LJOB0100, one or more selected fields are returned. They are defined by special identifiers, such as "0302" for a message with replacement data. The user must specify their number and offset to these fields.

The last field in format LJOB0100 specifies the name of the call message queue from which the messages are listed. You must use one of these values:

*****	Messages from every call stack entry of the job are listed.
***EXT**	Only messages sent to the external message queue (*EXT) of the job are to be listed.

Format of generated lists

To provide a consistent design and use of the user space (*USRSPC) objects, the OS/400 list APIs use a common data structure. The list APIs are those APIs that generate a list and have a user space parameter, such as the List Job Log Messages (QMHLJOBL). The user space created with QMHLJOBL API consists of the following areas:

- A user area
- A generic header
- An input parameter section
- A header section
- LJOB0100 format

Figure 24 on page 180 shows the general data structure for list APIs.

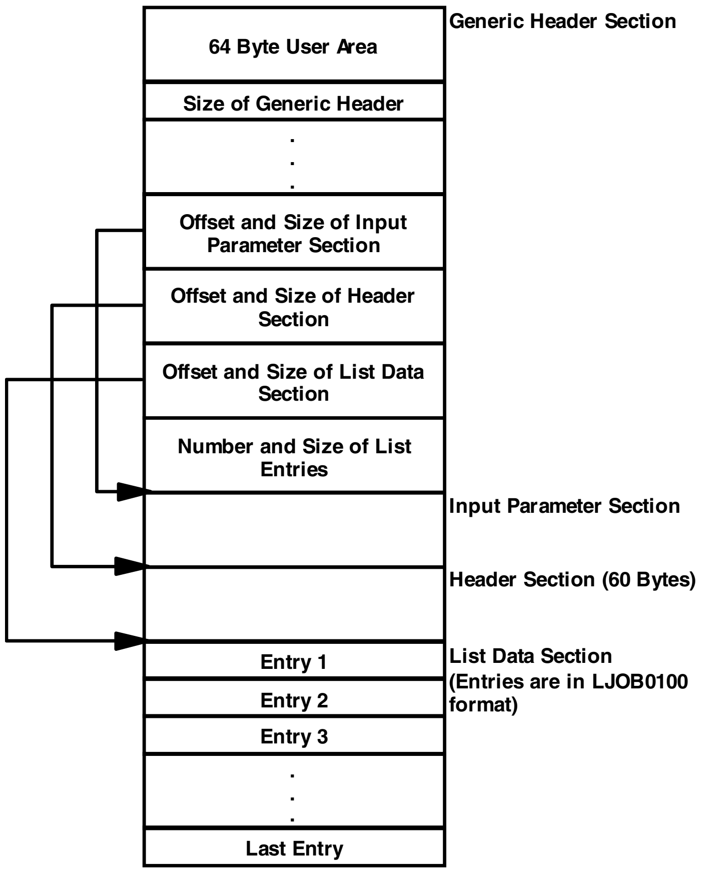

64 Byte User Area
Size of Generic Header
Offset and Size of Input Parameter Section
Offset and Size of Header Section
Offset and Size of List Data Section
Number and Size of List Entries
Entry 1
Entry 2
Entry 3
Last Entry
Generic Header Section
Input Parameter Section
Header Section (60 Bytes)
List Data Section (Entries are in LJOB0100 format)

Figure 24. General data structure for list APIs

Generic header format

Table 44 shows the layout of the generic header format for an original program model (OPM) program.

Table 44. Generic header format

From	To	Description	Type
1	64	User area	Char(64)
65	68	Size of generic header	Integer(10)
69	72	Structure's release and level	Char(4)
73	80	Format name	Char(8)
81	90	API used	Char(10)
91	103	Date and time created	Char(13)
104	104	Information status	Char(1)
105	108	Size of user space used	Integer(10)
109	112	Offset to input parameter section	Integer(10)
113	116	Size of input parameter section	Integer(10)
117	120	Offset to header section	Integer(10)
121	124	Size of header section	Integer(10)

From	To	Description	Type
125	128	Offset to list data section	Integer(10)
129	132	Size of list data section	Integer(10)
133	136	Number of list entries	Integer(10)
137	140	Size of each entry	Integer(10)
141	144	CCSID of data in the list entries	Integer(10)

Input parameter section format

This section contains an exact copy of all the parameters coded in the call to the API. Table 45 shows the layout of the input parameter section format.

Table 45. Input parameter section format

From	To	Description	Type
1	10	User space name specified	Char(10)
11	20	User space library specified	Char(10)
21	28	Format name specified	Char(8)
29	36	Format of message selection information specified	Char(8)
37	40	Size of message selection information specified	Integer(10)
41	44	Maximum messages requested specified	Integer(10)
45	54	List direction specified	Char(10)
55	64	Job name specified	Char(10)
65	74	User profile specified	Char(10)
75	80	Job number specified	Char(6)
81	96	Internal job identifier specified	Char(16)
97	100	Staring message key specified	Char(4)
101	104	Maximum message length specified	Integer(10)
105	108	Maximum message help length specified	Integer(10)
109	112	Offset to identifiers of fields to return specified	Integer(10)
113	116	Number of fields to return specified	Integer(10)
117	120	Offset to call message queue specified	Integer(10)
121	124	Length of call message queue specified	Integer(10)
125	128	Coded character set identifier specified	Integer(10)
129	*	Reserved	Char(*)
The offsets to these fields are specified in the previous offset variables.		Identifiers of fields to return specified	Array(*) of Integer(10)
		Call message queue specified	Char(*)

Header section format

This section contains information about the current values of parameters used by this invocation of the API. Table 46 shows the layout of the header section format.

Table 46. Header section format

From	To	Description	Type
1	10	User space name used	Char(10)
11	20	User space library used	Char(10)
21	24	Starting message key used	Char(4)
25	28	Ending message key used	Char(4)
29	38	Job name used	Char(10)
39	48	User profile used	Char(10)
49	54	Job number used	Char(6)
55	56	Reserved	Char(2)
57	60	Coded character set identifier used	Integer(10)

LJOB0100 format

Table 47 shows the information returned in the list data section of the user space for the LJOB0100 format. The offsets listed are from the beginning of the user space. The structure defined by this format is repeated for each message entry returned.

Table 47. LJOB0100 format

From	To	Description	Type
1	4	Offset to the next entry	Integer(10)
5	8	Offset to fields returned	Integer(10)
9	12	Number of fields returned	Integer(10)
13	16	Message severity	Integer(10)
17	23	Message identifier	Char(7)
24	25	Message type	Char(2)
26	29	Message key	Char(4)
30	39	Message file name	Char(10)
40	49	Message file library specified at send time	Char(10)
50	56	Date sent	Char(7)
57	62	Time sent	Char(6)
63	*	Reserved	Char(*)

From	To	Description	Type
These fields repeat for each identifier specified.		Offset to the next field information returned	Integer(10)
		Length of field information returned	Integer(10)
		Identifier field	Integer(10)
		Type of data	Char(1)
		Status of data	Char(1)
		Reserved	Char(14)
		Length of data	Integer(10)
		Data	Char(*)
		Reserved	Char(*)

5.4.3 Programming with message handling APIs

Message handling APIs are used in an RPG IV program the same way as other OPM APIs. They can be used to establish communication between programs by sending messages instead of transferring program parameters. This technique is similar to the use of data queues for program communication, but is limited to the programs within the same job.

For the sake of simplicity, we use immediate messages in our examples instead of predefined messages, which must be defined in the message file.

5.4.3.1 Source code for prototypes MSGPROTO

Source member MSGPROTO contains prototypes for calling the message handling APIs QMHSNDPM, QMHRCVPM, and QMHLJOBL.

```
 * Filename MSGPROTO from APISRC in RPGISCOOL

 * Prototype for API QMHSNDPM - Send Program Message
D SendMessage     Pr                  ExtPgm('QMHSNDPM')            1
D  MsgId                         7A
D  QualMsgFile                  20A
D  MsgData                      30A
D  MsgDataLen                   10I 0 Const
D  MsgType                      10A   Const
D  CallStEntry                  10A
D  CallStCount                  10I 0
D  MsgKey                        4A
D  Error                        16A

 * Prototype for API QMHRCVPM - Receive Program Message
D RcvMessage      Pr                  ExtPgm('QMHRCVPM')            2
D  MsgData                      78A
D  MsgDataLen                   10I 0 Const
D  FormatName                    8A   Const
D  CallStEntry                  10A
D  CallStCount                  10I 0
D  MsgType                      10A   Const
D  MsgKey                        4A
D  WaitTime                     10I 0
D  MsgAction                    10A   Const
D  Error                        16A

 * Prototype for API QMHLJOBL - List Job Log Messages
D ListJobLog      Pr                  ExtPgm('QMHLJOBL')            3
D  QualUserSpc                  20A
D  FormatRet                     8A   Const
D  SelInfo                      84A
D  SelInfoSize                  10I 0 Const
```

```
D  FormatSel                     8A   Const
D  Error                        16A
```

Program notes

1 Prototype for calling system API QMHSNDPM. Two parameters, defined with keyword CONST, are passed as constant values.

2 Prototype for calling system API QMHRCVPM. Four parameters, defined with keyword CONST, are passed as constant values.

3 Prototype for calling system API QMHLJOBL. Three parameters, defined with keyword CONST, are passed as constant values.

5.4.3.2 Message handling example 1

In this example, we use two simple programs to demonstrate the program communication by exchanging messages. The program SNDMSG sends several messages to the program message queue of its calling program. Another program, RCVMSG, first calls the program SNDMSG. Then, it retrieves all messages from its program message queue and check, if they are sent from program SNDMSG.

Source code for the program SNDMSG

This program uses the API QMHSNDPM to send several messages to its caller:

```
 *  Filename SNDMSG from APISRC in RPGISCOOL

 * Include prototypes
D/Copy RPGISCOOL/APISRC,MSGPROTO                                    1

 * Copy QUSEC data structure for error handling
D/COPY QSYSINC/QRPGLESRC,QUSEC                                      2

 * Program variable definitions
D  MsgId          S              7A   Inz                           3
D  QualMsgFile    S             20A   Inz
D  MsgData        S             30A   Inz
D  CallStEntry    S             10A   Inz('*')
D  CallStCount    S             10I 0 Inz(2)
D  MsgKey         S              4A
D  Index          S              5I 0

C                   For       Index = 1 By 1 To 5

C                   Eval      MsgData = 'SESAME ' + %Char(Index) +
C                             '. Message from SNDMSG'
 * Call QMHSNDPM to send five messages to calling program           4
C                   CallP     SendMessage(MsgID : QualMsgFile : MsgData
C                             : %Len(MsgData) : '*INFO' : CallStEntry
C                             : CallStCount : MsgKey : QUSEC)

C                   EndFor

C                   Eval      *InLR = *On
```

SNDMSG program notes

1 Copy prototype definitions from member MSGPROTO.

2 Copy statement used to include a common structure for an error code parameter from library QSYSINC.

3 Parameter definitions with initial values. Call stack entry and call stack count parameters define the program to whom messages are sent. In this example, we do not use predefined messages, so the message ID and message file parameters are initialized to blanks.

4 Call the API to send a prepared message as an informational message (type *INFO). The program prepares the text for the impromptu message. The program marks it with the word "SESAME" in the first six positions, which will be used by the receiving program to identify these messages among others in its program message queue. This is necessary because we do not have a message identifier for impromptu messages.

Source code for the program RCVMSG

This program first calls the program SNDMSG, which sends several messages. Then, it uses the API QMHRCVPM to retrieve all these messages from its program message queue.

```
 *  Filename RCVMSG from APISRC in RPGISCOOL

 * Include prototypes
D/Copy RPGISCOOL/APISRC,MSGPROTO                                     1

 * Copy QUSEC data structure for error handling
D/COPY QSYSINC/QRPGLESRC,QUSEC                                       2

 * Format RCVM0100 with received fields
D RCVM01DS        DS                                                 3
D  BytesRetrn                    10I 0
D  BytesAvlbl                    10I 0
D  MsgSev                        10I 0
D  MsgId                          7A
D  MsgTypeRet                     2A
D  MsgKeyRet                      4A
D  Reserved                       7A
D  CCSID1                        10I 0
D  CCSID2                        10I 0
D  DtaLenRetrn                   10I 0
D  DtaLenAvlbl                   10I 0
D* DtaLenAvlbl                   10I 0 Inz(30)
D  MsgTextRet                    30A

 * Program variable definitions
D  CallStEntry    S              10A   Inz('*')                      4
D  CallStCount    S              10I 0 Inz(0)
D  MsgKey         S               4A
D  Wait           S              10I 0 Inz(0)

C                   Call      'SNDMSG'                               5

C                   DoW       not *InLR

 * Call QMHRCVPM to receive all messages sent to this program
C                   CallP     RcvMessage(RCVM01DS : %Len(RCVM01DS) :   6
C                             'RCVM0100' : CallStEntry : CallStCount :
C                             '*INFO' : MsgKey : Wait : '*REMOVE' :
C                             QUSEC)
C                   If        BytesRetrn=8
C                   Leave
C                   EndIf

 * Check if the message was sent from SNDMSG program
C                   If        %Subst(MsgTextRet : 1 : 6) = 'SESAME'    7
C     MsgTextRet    Dsply
C                   EndIf

C                   EndDo

C                   Eval      *InLR = *On
```

RCVMSG program notes

1 Copy prototype definitions from member MSGPROTO.

2 Copy statement used to include the common structure for the error code parameter from the QSYSINC library.

3 Data structure that maps format RCVM0100 fields with all values received from the message.

4 Parameter definitions with initial values. Call stack entry and call stack count parameters define that the messages for the current program are to be read.

5 Call program SNDMSG to produce messages.

6 Call the API to receive the message and put its data into the RCVM0100 format. Due to the *REMOVE action, a received message is removed from the message queue. Check for no more messages condition by comparing bytes returned value to 8.

7 To identify messages sent from the program SNDMSG, check for the word "SESAME" in the first six positions of the message data. This is used as a replacement for the message ID.

Try it yourself

You can try this example by compiling the code from this section on your AS/400 system. Use the following command to create the programs:

```
CRTBNDRPG PGM(RPGISCOOL/SNDMSG) SRCFILE(RPGISCOOL/APISRC)
CRTBNDRPG PGM(RPGISCOOL/RCVMSG) SRCFILE(RPGISCOOL/APISRC)
```

To run the program, use the following command:

```
CALL PGM(RPGISCOOL/RCVMSG)
```

You get the following answers on your screen:

```
SESAME 1. Message from SNDMSG
SESAME 2. Message from SNDMSG
SESAME 3. Message from SNDMSG
SESAME 4. Message from SNDMSG
SESAME 5. Message from SNDMSG
```

5.4.3.3 Message handling example 2

In this example, we use the already known program SNDMSG to send several messages to the program message queue of its calling program. The LSTMSG calling program uses another technique this time to receive messages. Invoking the API QMHLJOBL, it gets all messages from the job log listed in the user space, and then reads them by checking if they are sent from the program SNDMSG.

There is an example of a practical use for user spaces described in 5.3, “User space APIs” on page 153.

Source code for the program LSTMSG

This program uses the API QMHLJOBL to get the list of all messages from the job log in the user space. To define all the data structures needed by this API, we could use copy member QMHLJOBL from the QRPGLESRC file in the library QSYSINC, but it contains short and meaningless field names. We have decided to define our own data structures with meaningful names for better readability of the program.

To create a user space and to retrieve the pointer to this user space, in this example, we use two procedures created in 5.3, “User space APIs” on page 153. They are contained in the program SPACEPROCS.

The source program must be compiled into the module and then bound with the module SPACEPROCS to create an executable program.

```
 *  Filename LSTMSG from APISRC in RPGISCOOL

 * Include prototypes
D/Copy RPGISCOOL/APISRC,MSGPROTO                                              1
D/Copy RPGISCOOL/APISRC,SPACEPROTO                                            2

 * Copy QUSEC data structure for error handling
D/COPY QSYSINC/QRPGLESRC,QUSEC                                                3

 * Format JSLT0100 with selection parameter
D JSLT01DS        DS                                                          4
D  MaxMsgReq                    10I 0 Inz(-1)
D  ListDir                      10A   Inz('*NEXT')
D  JobName                      10A   Inz('*')
D  UserName                     10A   Inz
D  JobNumber                     6A   Inz
D  IntJobId                     16A   Inz
D  StrMsgKey                     4A   Inz(X'00000000')
D  MaxMsgLen                    10I 0 Inz(-1)
D  MaxMsgHLen                   10I 0 Inz
D  OffsetFldRet                 10I 0 Inz(84)
D  NumFldRet                    10I 0 Inz(1)
D  OffsetMsgQ                   10I 0 Inz(88)
D  LengthMsgQ                   10I 0 Inz(4)
D  Field1                       10I 0 Inz(0302)
D  MQName                        4A   Inz('*')

 * Generic Header data structure
D GenHeadDS       DS                  Based(BasePtr)                          5
D  UserArea                     64
D  SizeGenHed                   10I 0
D  StrRelLvl                     4A
D  FmtName                       8A
D  APIName                      10A
D  DateTime                     13A
D  InfStatus                     1A
D  UserSpcSize                  10I 0
D  IPSecOffset                  10I 0
D  IPSecSize                    10I 0
D  HSecOffset                   10I 0
D  HSecSize                     10I 0
D  LDSecOffset                  10I 0
D  LDSecSize                    10I 0
D  EntryNumber                  10I 0
D  EntrySize                    10I 0
D  CCSID                        10I 0

 * Format LJOB0100 with message entry data
D LJOB01DS        DS                  Based(ListPtr)                          6
D  NxtEntOffset                 10I 0
D  FldRetOffset                 10I 0
D  FldRetNumber                 10I 0
D  MsgSev                       10I 0
D  MsgID                         7A
D  MsgType                       2A
D  MsgKey                        4A
D  MsgFileName                  10A
D  MsgFileLib                   10A
D  DateSent                      7A
D  TimeSent                      6A
D  Reserved1                  8192A

 * Message entry format at end of LJOB0100 format
D MsgEntryDS      DS                  Based(EntryPtr)                         7
D  NxtFldOffset                 10I 0
D  FldInfLen                    10I 0
D  FieldID                      10I 0
D  DataType                      1A
D  DataStatus                    1A
D  Reserved2                    14A
D  DataLen                      10I 0
D  DataText                     30A

 * Program variable definitions
D QualUserSpc     DS
```

```
D  SpaceName                    10A   Inz('JOBLOG')
D  Library                      10A   Inz('*CURLIB')

C                   Call      'SNDMSG'                              8

 * Call QUSCRTUS to create user space
C                   CallP     CreateSpace(SpaceName)                9
 * Call QUSPTRUS to retrieve pointer to user space
C                   Eval      BasePtr=GetSpacePtr(SpaceName:Library)  10
 * Call QHMLJOBL to list job log messages
C                   CallP     ListJobLog(QualUserSpc : 'LJOB0100'    11
C                             : JSLT01DS : %Len(JSLT01DS) : 'JSLT0100'
C                             : QUSEC)
C                   Eval      ListPtr = BasePtr + LDSecOffset        12
C                   Eval      EntryPtr = BasePtr + FldRetOffset

C                   Do        EntryNumber

C                   If        %Subst(DataText : 1 : 6) = 'SESAME'   13
C     DataText      Dsply
C                   EndIf
C                   Eval      ListPtr = BasePtr + NxtEntOffset
C                   Eval      EntryPtr = BasePtr + FldRetOffset

C                   EndDo

C                   Eval      *InLR = *On
```

LSTMSG program notes

1 Copy prototype definitions from member MSGPROTO.

2 Copy prototype definitions from member SPACEPROTO.

3 Copy statement used to include the common structure for the Error code parameter from the library QSYSINC.

4 Data structure which maps format JSLT0100 fields, where we define all message selection criteria. In our case, we request only the data part (field identifier 0302) to be retrieved from all messages in the job log of the current job.

5 Data structure that maps generic header fields at the beginning of the user space. This data structure contains fields with offsets and sizes of other sections in the generated list. It is based on the pointer that is loaded by the GetSpacePtr procedure.

6 Data structure that maps format LJOB0100 fields, where message entry data is returned. For each message retrieved from the job log, one entry in this format is returned in the list data section. It is based on another pointer to allow scrolling through all message entries.

7 Data structure that maps requested fields from one message entry. In our case, we have only one field containing message data. It is also based on pointer to allow scrolling through all requested fields.

8 Call program SNDMSG to produce messages.

9 Procedure CreateSpace creates new user space.

10 Procedure GetSpacePtr loads BasePtr with the address of the user space.

11 Call system API QMHLJOBL to list selected job log entries into the user space.

12 Load pointers needed to access message data in the user space.

13 In the loop controlled by the number of entries, check first six characters of the message data to identify entries sent from program SNFMSG. Then, increase the pointers to access the next entry.

Try it yourself

You can try this example by compiling the code from this section on your AS/400 system. Use the following commands to create the programs:

```
CRTRPGMOD MODULE(RPGISCOOL/LSTMSG) SRCFILE(RPGISCOOL/APISRC)
CRTPGM PGM(RPGISCOOL/LSTMSG) MODULE(LSTMSG SPACEPROCS)
```

To run the program, use the following command:

```
CALL PGM(RPGISCOOL/LSTMSG)
```

The following answers appear on your screen:

```
SESAME 1. Message from SNDMSG
SESAME 2. Message from SNDMSG
SESAME 3. Message from SNDMSG
SESAME 4. Message from SNDMSG
SESAME 5. Message from SNDMSG
```

5.5 Sockets

Sockets are traditionally thought of as being in the domain of the C programmer. This belief is reinforced by some IBM documentation, which suggests that you cannot use sockets functions from RPG. Luckily for you, we did not read those parts of the manuals before writing these examples!

If you find yourself interested in sockets programming, but do not understand the background, please review *OS/400 Sockets Programming V4R4*, SC41-5422. This manual offers a good discussion of socket programming, in general, and on the AS/400 system. It does not discuss RPG, which is the purpose of this section.

The examples in this section show two programs: client and server. The server starts and waits for messages from the client. The client initiates the communication by sending a message to the server. In our examples, the client is an interactive program that sends a request to the server and receives the answer from the server. The server can be run in batch or interactive mode. The examples all use connection-oriented Socket communication. This type of communication is the most commonly used due to its reliability.

Both the server and client programs are written in RPG IV. A more likely situation is when one of the programs is written and run in another platform, typically in the C language. Therefore, the RPG IV programmer can use either side of the socket communication from our examples. When using sockets with other systems, the programmer has to translate data from ASCII to EBCDIC code and vice versa.

5.5.1 Typical communication between a server and client

"Connection-oriented" implies that a connection is established and a dialog between the programs will follow. The program that provides the service (the server program) establishes the connection and gives itself a name of where that service can be obtained. The client of the service (the client program) must request the service of the server program by connecting to the distinct name that the server program has designated. It is like you (a client) dialing a telephone number (an identifier) and making a connection with another party offering a service (for example, a plumber). Once the receiver of the call (the server) answers the telephone, the connection is established. The caller can verify that they reached the correct party and the connection remains active as long as both parties require it.

The following socket functions are most commonly used in the connection-oriented communication:

socket	Obtains a new socket description (both client and server).
setsockopt	Sets various attributes to the socket (both server and client).
bind	Binds the program to the socket (server).
listen	Enables the client program to connect the server program (server).
accept	Waits for the connect function from the client program (server).
connect	Connects to the server (client).
gethostbyname	Transforms the host name to an IP address (client).
inetaddr	Transforms an IP address from a dotted to a binary form.
read	Reads a message from the socket. The *recv* function can be used as well.
write	Writes a message to the socket. The *send* function can be used as well.
close	Closes the socket (both client and server).

Figure 25 shows how a typical communication between two programs appears.

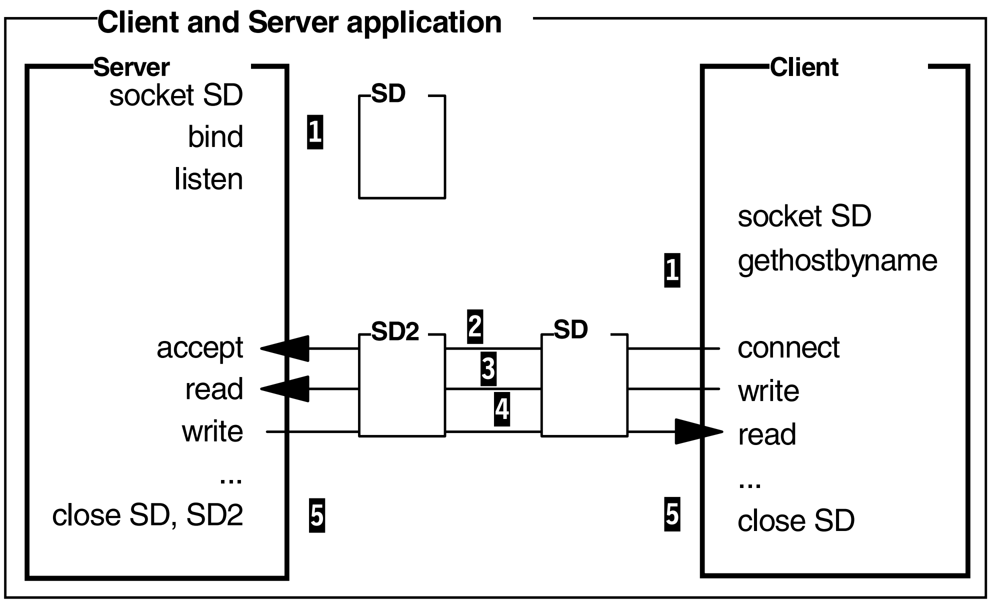


Figure 25. Communication between the server and client

The communication process in Figure 25 is described in the following series of events:

1. The server is started first and makes itself available for the communication by issuing *socket*, *bind*, and *listen* functions using its own socket description SD.
2. The server issues the *accept* function that waits for a connect request from the client.
3. The client is started and issues the *socket* and *gethostbyname* (or *inetaddr*) functions using its own socket description SD **1**.
4. The client issues a *connect* request that is accepted by the server. The accept function in the server creates a second socket description SD2. The server uses it for further communication with the client **2**.
5. The server reads data from the client using the *read* function through the second socket description SD2 (or waits if no data comes) **3**.
6. The client sends the first data using the *write* function over its SD socket. The server receives it by using the *read* function through the socket description SD2 **3**.
7. The server processes the data and sends a reply to the client using the *write* function **4**.
8. The client receives the reply by using the *read* function and processes it **4**.
9. Communication proceeds by alternating write/read functions on the client side and read/write functions on the server side.
10. The communication can be ended by the client that can send special request data telling the server to finish its work. Both the client and the server issue the *close* function. The server closes the SD and SD2 sockets, and the client closes its SD socket **5**.

5.5.2 The socket functions interface

The socket functions used in the following examples are described along with their interface, including the prototypes and the special values.

The RPG IV function prototypes are presented both in the C language and RPG IV language in the form as they are entered in the CSKCPY /COPY member.

The C prototypes are contained in the QSYSINC library, H source file. There are no QSYSINC prototypes for RPG IV socket functions for the time being.

5.5.2.1 The socket() function

The socket function is used to create an endpoint for communications. The endpoint is represented by the socket descriptor returned by the socket function. The *socket descriptor* is a structure, also called the *socket address*, and is described at the bind function.

```
 *   int socket(int address_family,
 *              int type,
 *              int protocol)

D Socket          Pr            10I 0 Extproc('socket')

D                               10I 0 Value
D                               10I 0 Value
D                               10I 0 Value
```

Table 48 shows the parameters for this function.

Table 48. Parameters for the socket() function

Argument	Description	Use	RPG IV data type	C data type
address_family	Address family	Input	Integer(10)	int
type	Communication type	Input	Integer(10)	int
protocol	Protocol used with sockets	Input	Integer(10)	int

The return values are:

n	Socket descriptor number
-1	If unsuccessful

The *address_family* parameter can have the following values:

AF_INET Interprocess communications between processes on the same system or different systems in the Internet domain. Its decimal value is 2. We use this value in all our examples.

AF_NS Interprocess communications between processes on the same system or different systems in the domain defined by the Novell or Xerox protocol definitions. Its decimal value is 6.

AF_UNIX Interprocess communications between processes on the same system in the UNIX domain. Its decimal value is 1.

AF_TELEPHONY
Interprocess communications between processes on the same system in the telephony domain. Its decimal value is 99.

These values are defined in the CSKCPY /COPY member.

The *type* parameter can have the following values:

SOCK_STREAM
Indicates that a full-duplex stream socket is desired. We use this value in all our examples. Its decimal value is 1.

SOCK_DGRAM
Indicates that a datagram socket is desired. Its decimal value is 2.

SOCK_SEQPACKET
Indicates that a full-duplex sequenced packet socket is desired. Each input and output operation consists of exactly one record. Its decimal value is 5.

SOCK_RAW
Indicates that communication is directly to the network protocols. A process must have the appropriate privilege *ALLOBJ to use this type of socket. Used by users who want to access the lower-level protocols directly. Its decimal value is 3.

These values are defined in the CSKCPY /COPY member.

The *protocol* parameter can have many values of which 0 (zero) designates the default protocol for the address family and type parameters.

5.5.2.2 The setsockopt() function

The setsockopt() function is used to set various socket options. We use this function to set a socket to be reusable. It means that the local socket address can be reused.

```
 *   int setsockopt(int socket_descriptor,
 *                  int level,
 *                  int option_name,
 *                  char *option_value,
 *                  int option_length)
 *
D SetsockOpt      Pr            10I 0 Extproc('setsockopt')
D                               10I 0 Value
D                               10I 0 Value
D                               10I 0 Value
D                                 *   Value
D                               10I 0 Value
```

Table 49 shows the parameters for this function.

Table 49. Parameters for the setSockOpt() function

Argument	Description	Use	RPG data type	C data type
socket_descriptor	Socket descriptor number	Input	Integer(10)	int
level	We use SOL_SOCKET (-1)	Input	Integer(10)	int
option_name	We use SO_REUSADDR (55)	Input	Integer(10)	int
option_value	We use pointer to 1	Input	Pointer	char *
option_length	Length of the option value	Input	Integer(10)	int

The return values are:

0	Successful
-1	Unsuccessful

5.5.2.3 The bind() function

The bind() function is used to associate a local address with a socket.

```
 *   int bind(int socket_descriptor,
 *            struct sockaddr *local_address,
 *            int address_length)

D Bind            Pr            10I 0 ExtProc('bind')

D                               10I 0 Value
D                                 *   Value
D                               10I 0 Value
```

Table 50 on page 194 shows the parameters for this function.

Table 50. Parameters for the bind() function

Argument	Description	Use	RPG data type	C data type
socket_descriptor	Socket descriptor number	Input	Integer(10)	int
local_address	Pointer to local address structure	Input	Pointer	struct *
address_length	Length of local address	Input	Integer(10)	int

The return values are:

0	Successful
-1	Unsuccessful

The local address structure for the AF_INET address family has the following format in the C language:

```
struct sockaddr_in {          /* socket address (internet)   */
   short sin_family;          /* address family (AF_INET)    */
   u_short sin_port;          /* port number                 */
   struct in_addr  sin_addr;  /* IP address                  */
   char sin_zero[8];          /* reserved - must be 0x00's   */
};
```

The structure in RPG IV looks like this:

```
D SocketAddr      DS
D   SinFamily                     5I 0
D   SinPort                       5U 0
D   SinAddr                      10U 0
D   SinZero                       8A   Inz( X'00' )
```

The IP address in binary form is placed in the SinAddr field.

5.5.2.4 The listen() function

The listen() function is used to indicate a willingness to accept incoming connection requests. If a listen() is not done, incoming connections are silently discarded.

```
 *   int listen(int socket_descriptor,
 *              int back_log)

D Listen          Pr            10I 0 ExtProc('listen')

D                               10I 0 Value
D                               10I 0 Value
```

Table 51 shows the parameters for this function.

Table 51. Parameters for the listen() function

Argument	Description	Use	RPG data type	C data type
socket_descriptor	Socket descriptor number	Input	Integer(10)	int
back_log	Number of clients to be accepted	Input	Integer(10)	int

The return values are:

0	Successful
-1	Unsuccessful

5.5.2.5 The accept() function

The accept() function is used to wait for connection requests. The accept() function takes the first connection request on the queue of the pending connection requests and creates a new socket to service the connection request. The accept function is used with connection-oriented socket types, such as SOCK_STREAM.

```
 *   int accept(int socket_descriptor,
 *              struct sockaddr *address,
 *              int *address_length)

D Accept          Pr            10I 0 ExtProc('accept')

D                               10I 0 Value
D                                 *   Value
D                                 *   Value
```

Table 52 shows the parameters for this function.

Table 52. Parameters for the accept() function

Argument	Description	Use	RPG data type	C data type
socket_descriptor	Socket descriptor number	Input	Integer(10)	int
address	Pointer to socket address structure	Input	Pointer	struct *
address_length	Length of socket address	Input	Integer(10)	int

The return values are:

0	Successful
-1	Unsuccessful

The socket address structure is described in the bind function.

5.5.2.6 The connect() function

The connect() function is used to establish a connection on a connection-oriented socket or establish the destination address on a connectionless socket.

```
 *   int connect(int socket_descriptor,
 *               struct sockaddr *destination_address,
 *               int address_length)

D Connect         Pr            10I 0 Extproc('connect')
D                               10I 0 Value
D                                 *   Value
D                               10I 0 Value
```

Table 53 shows the parameters for this function.

Table 53. Parameters for the connect() function

Argument	Description	Use	RPG data type	C data type
socket_descriptor	Socket descriptor number	Input	Integer(10)	int
destination_address	Pointer to socket address structure	Input	Pointer	struct *
address_length	Length of socket address	Input	Integer(10)	int

The return values are:

0	Successful
-1	Unsuccessful

The socket address structure is as described in 5.5.2.3, "The bind() function" on page 193.

5.5.2.7 The gethostbyname() function

The gethostbyname() function is used to retrieve information about a host. In this case, it transforms the host name into the binary coded IP address. It returns a pointer to the host entry structure. You use this function to obtain the IP address for a hostname. The gethostbyname() function can take quite a long time before it finds the IP address in appropriate tables.

```
 *   struct HostEnt *GetHostByName(char *host_name)

D GetHostByName   Pr               *   Extproc('gethostbyname')

D                                  *   Value
```

The function has only one parameter, a pointer to the host name.

The return value is a pointer to the host entry structure, which is, in fact, a hierarchy of structures. We use only one path in the hierarchy to find the IP address.

The structure has the following form in the C language:

```
 *   struct HostEnt {
 *      char   *h_name;
 *      char   **h_aliases;
 *      int    h_addrtype;
 *      int    h_length;
 *      char   **h_addr_list;
 *   };
```

The corresponding structure in the RPG IV language is as follows:

```
D HostEnt         DS                   Align Based(Host@)
D  HName@                          *
D  HAliases@                       *
D  HAddrType                     10I 0
D  HLength                       10I 0
D  HAddrList@                      *
```

`HName@` is a pointer that points to the host entry data structure where the IP address is stored.

The host entry data structure has this form in the C language:

```
struct hostent_data        {          /* additional host entry data
                                         Considered opaque. Must be
                                         16 byte aligned. */
  char  h_name[NETDB_MAX_HOST_NAME_LENGTH+1];
                                      /* host name */
  char  *h_aliases_arrayp[NETDB_MAX_ARRAY_SIZE+1];
                                      /* Array of pointers to
                                         h_aliases_array elements */
  char h_aliases_array[NETDB_MAX_ARRAY_SIZE]
                           [NETDB_MAX_HOST_NAME_LENGTH+1];
                                      /* Alias Array */
  char *h_addr_arrayp[NETDB_MAX_HOST_ADDR_ARRAY_SIZE+1];
                                      /* Array of pointers to
                                         h_addr_array elements */
  struct in_addr h_addr_array[NETDB_MAX_HOST_ADDR_ARRAY_SIZE];
                                     /* Host address array */
  struct netdb_control_block host_control_blk;
                   };
```

This structure has the following form in the RPG IV language:

```
D HostEntData     DS                  Align Based(HostEntData@)
D  HName                       256A
D  HAliasesArr@                  *    Dim(65)
D  HAliasesArr                 256A   Dim(64)
D  HAddrArr@                     *    Dim(101)
D  HAddrArr                     10U 0 Dim(100)
D  OpenFlag                     10I 0
D  F0@                           *
D  FileP0                      260A
D  Reserved0                   150A
D  F1@                           *
D  FileP1                      260A
D  Reserved1                   150A
D  F2@                           *
D  FileP2                      260A
D  Reserved2                   150A
```

We use the path "`HName@` - `HAddrArr@`" to find the IP address. `HAddrArr@` points to an array of IP addresses of which we need the first address.

5.5.2.8 The InetAddr() function

The inet_addr function is used to transform the IP address in the dotted form into its binary representation. It returns an unsigned integer (four bytes long) containing the binary IP address.

```
 *-- InetAddr --- Transform IP address from dotted form ----------
 *   unsigned long inet_addr(char *address_string)
D InetAddr        Pr            10U 0 ExtProc('inet_addr')
D                                 *   Value
```

The function has only one parameter, a character value containing the IP address in the dotted decimal form. Notation of the dotted decimal IP address value can be in one of seven formats:

```
Format 1 - a.b.c.d
Format 2 - a.b.c.
Format 3 - a.b.c
Format 4 - a.b.
Format 5 - a.b
Format 6 - a.
Format 7 - a
```

The rules for converting a dotted decimal string are as follows:

- For format 1, each component is interpreted as one byte of the Internet address.
- For format 2, each component is interpreted as one byte of the Internet address. The right-most byte is set to zero.
- For format 3, each component is interpreted as one byte of the Internet address, except for component c, which is interpreted as the right-most two bytes of the Internet address.
- For format 4, each component is interpreted as one byte of the Internet address. The right-most two bytes are set to zero.
- For format 5, each component is interpreted as one byte of the Internet address, except for component b, which is interpreted as the right-most three bytes of the Internet address.
- For format 6, component a is interpreted as one byte of the Internet address. The right-most three bytes are set to zero.
- For format 7, component a is returned as the Internet address.

The return values are:

-1	Unsuccessful
n	The 32-bit IP address

5.5.2.9 The read() function

The read() function reads data of the length specified in *buffer_length* from the input into the memory area indicated by the *buffer*. If the *buffer_length* is zero, the read() function returns a value of zero without attempting any other action.

```
 *   ssize_t read(int descriptor,
 *                void *buffer,
 *                size_t buffer_length)

D Read            Pr            10I 0 Extproc('read')

D                               10I 0 Value
D                                 *   Value
D                               10U 0 Value
```

Table 54 shows the parameters for this function.

Table 54. Parameters for the read() function

Argument	Description	Use	RPG data type	C data type
socket_descriptor	Socket descriptor number	Input	Integer(10)	int
buffer	Pointer to a data buffer	Input	Pointer	void *
buffer_length	Length of data to be read	Input	Integer(10)	int

The return values are:

n	The read function *was* successful. The value returned is the number of bytes actually read and placed in the *buffer*. The *errno* global variable is set to a nonzero value to indicate the error.

0 The read function *was not* successful. The partner's socket is closed. The *errno* global variable is set to zero to indicate no error.

-1 The read function *was not* successful. The *errno* global variable is set to indicate the error.

5.5.2.10 The write() function

The write() function writes data of the length specified in *buffer_length* from the *buffer* to the socket. If the *buffer_length* is zero, write() returns a value of zero without attempting any other action. The write() function can write data also to files because files also use descriptors.

```
 *   ssize_t write (int file_descriptor,
 *                  const void *buffer
 *                  size_t buffer_length);

D Write           Pr            10I 0 ExtProc('write')

D                               10I 0 VALUE
D                                 *   VALUE
D                               10U 0 VALUE
```

Table 55 shows the parameters for this function.

Table 55. Parameters for the write() function

Argument	Description	Use	RPG data type	C data type
socket_descriptor	Socket descriptor number	Input	Integer(10)	int
buffer	Pointer to a data buffer	Input	Pointer	void *
buffer_length	Length of data to be written	Input	Integer(10)	int

The return values are:

n The write function *was* successful. The value returned is the number of bytes actually written. This number is equal to *buffer_length*.

-1 The write function *was not* successful. The errno global variable is set to indicate the error.

5.5.2.11 The close() function

The close() function closes the socket identified by the descriptor. We do not test the return value in our examples calling the close() function by the CALLP operation.

```
 *   int close(int descriptor)

D Close           Pr                  ExtProc('close')

D                               10I 0 Value
```

The return values are:

0 The close *was* successful.

-1 The close *was not* successful. The errno global variable is set to indicate the error.

5.5.3 Example of a simple server SSERVER and client SCLIENT

There are two programs: one server and one client. The server receives requests from the client and responses to the requests. The request data from the client is an item number that a user enters from the keyboard. The server finds the record in the ITEMS file and sends it back to the client. The client displays the record on the screen. If the client sends the item number "END", both the server and client end.

This example is quite simple. Errors are checked only for debugging purposes. The main socket functions are illustrated and some inconveniences of these simple programs are highlighted.

The programs SSERVER and SCLIENT are written according to Figure 25 on page 190.

5.5.3.1 The SSERVER program

To understand the logic of this program better, we break it up into the logical pieces of data definition, procedure, and error handling.

SSERVER program: Data definition

Here is the source listing of the SSERVER program:

```
H DFTACTGRP(*NO) ACTGRP('QILE') BNDDIR('QC2LE')                      1

 *   Database ITEMS file
FITEMS     IF   E           K DISK

D SocketData    E Ds                  ExtName(ITEMS)                 2
D SocketData@     S               *   Inz(%Addr(SocketData))
D SockDtaLen      S             10I 0 Inz(%size(SocketData))
D PortNumber      S             10I 0 Inz(3005)                      3
D SD              S             10I 0                                4
D SD2             S             10I 0                                5
D RC              S             10I 0                                6
D OptVal          S             10U 0 Inz(1)                         7

 *   Necessary procedrure prototypes with some data definitions
 /COPY SCKSRC,SCKCPY                                                  8

D ErrorHdlr       Pr                                                 9
D  DumpText                      12    Value
```

1 The server program runs in the activation group QILE and binds only the necessary service programs from those listed in binding directory QC2LE. These service programs contain common C language functions.

2 SocketData is a variable for data exchange between server and client. It is structured like the ITEMS file record. Its address and length are required in the socket *read* and *write* functions.

3 PortNumber is needed to bind the socket with an IP address. The variable is initialized by a port number that does not match with "well known" port numbers. The well-known port numbers are assigned to various applications. TCP and UDP protocols use ports to identify a unique origin or the destination of the communication with an application. Each port is assigned a short integer (5U 0 in RPG IV).

4 SD is a socket description number obtained by the server through the *socket* function. This socket is used by the server for listening to clients.

5 SD2 is a socket description number obtained by the server through the *accept* function. This socket is used for conversation between the client and the server.

6 RC serves as a general return code for socket functions.

7 OptVal is a nonzero number required by the *setsockopt* function, which is used here for setting a socket to be reusable.

8 The /COPY statement brings all necessary prototype definitions for socket functions (subprocedures) contained in service programs of the operating system.

9 The prototype for our own error handling subprocedure is specified.

SSERVER program: Procedure

The server obtains a socket, makes it reusable, listens to one client, binds to an IP address and a port, accepts the connection request from the client by creating a new socket, and enters a processing loop. In the loop, the server reads data from the client, processes it, and sends back a response to the client back. When the data is "END", the server closes sockets and ends.

```
 *   Obtain a socket descriptor for itself                             1
C                   Eval      SD = Socket (AF_INET: SOCK_STREAM: 0 )

 *   If socket failed - End the server program with dump
C                   If        SD < 0                                    2
C                   CallP     ErrorHdlr ('ServerSocket')
C                   Return
C                   EndIf

 *   Allow socket description to be reusable                            3
C                   Eval      RC = SetSockOpt (SD: SOL_SOCKET
C                                                 : SO_REUSEADDR
C                                                 : %Addr(OptVal)
C                                                 : %Size(OptVal)  )

 *   Bind the socket to an IP address                                   4
C                   Eval      SocketAddr = *ALLX'00'
C                   Eval      SinFamily  = AF_INET
C                   Eval      SinPort    = PortNumber
C                   Eval      SinAddr    = INADDR_ANY
C                   Eval      RC = Bind (SD: %ADDR(SocketAddr)          5
C                                           : %SIZE(SocketAddr))

 *   If bind failed - End the server program with dump
C                   If        RC < 0
C                   CallP     ErrorHdlr ('ServerBind')
C                   Return
C                   EndIf

 *   Listen to one client only
C                   Eval      RC = Listen (SD: 1)                       6

 *   If listen failed - End the server program with dump
C                   If        RC < 0
C                   CallP     ErrorHdlr ('ServerListen')
C                   Return
C                   EndIf

 *   Accept incoming connection request from the client.
 *   A new socket (SD2) is created for the client.
C                   Eval      SD2 = Accept (SD: SockAddr: AddrLen)      7

 *   If accept failed - End the server program with dump
C                   If        RC < 0
C                   CallP     ErrorHdlr ('ServerAccept')
C                   Return
C                   EndIf

 *   Enter read/write loop
```

```
C                   DoW       0 = 0

 *   Read data from the client's socket to SocketData variable         8
C                   Eval       RC = Read (SD2: SocketData@: SockDtaLen)

 *   If read failed - End the server program with dump
C                   If         RC <= 0
C                   CallP     ErrorHdlr ('ServerRead')
C                   EndIf

 *   If the first characters of item number are END - end the server
C                   If        ITEMNBR  = 'END'                         9
C                   Leave
C                   EndIf

 *   Read the corresponding record from the ITEMS file by key
C     ITEMNBR       Chain     ITEMS

 *   If found - Send data to the client
C                   If        Not %Found

 *   If not found - Send question marks to the client
C                   Eval      UNITPR = 0
C                   Eval      ITEMDESC = *All'?'                      10
C                   EndIf

 *   Write response to the client (the item record or question marks) 11
C                   Eval      RC = Write (SD2: SocketData@: SockDtaLen)

 *   If write failed - End the server program with dump
C                   If         RC <= 0
C                   CallP     ErrorHdlr ('ServerWrite')
C                   Return
C                   EndIf

 *   End read/write loop
C                   EndDo

 *   End program
C                   CallP     Close(SD2)                              12
C                   CallP     Close(SD)
C                   Eval      *InLR = *ON
```

1 The socket function creates a socket descriptor and returns its number in the SD variable.

2 An SD value of 0 or more represents a valid socket description number. If SD = -1 the socket function was not successful. In this case, an error handling subprocedure is called, which dumps the memory and ends the server program.

3 SetSockOpt is a function that sets various attributes to the socket. In this case, it indicates that the local socket address can be reused. The socket can be used again after it is closed.

4 Before the bind function can be performed, the socket local address data structure must be initialized by certain values. The socket local address has the following structure:

```
D SocketAddr      DS
D   SinFamily                    5I 0
D   SinPort                      5U 0
D   SinAddr                     10U 0
D   SinZero                      8A   Inz( *ALLX'00' )
```

The following values are used in socket functions as parameters:

```
D SockAddr        S               *   Inz( %Addr(SocketAddr) )
D AddressLength   S             10I 0
D AddrLen         S               *   Inz( %Addr(AddressLength) )
```

The numeric value of INADDR_ANY is 0.

5 The bind function returns 0 if successful, or -1if it is not successful.

6 The listen function prepares the server to communicate with one client.

7 The accept function waits for the client's connect request. When the request comes, the accept function returns a new socket description number in the SD2 variable. If SD2 is -1, the function ended with error.

8 Communication between the client and server occurs in an "infinite" loop. The server issues a read function over the SD2 socket and waits for a request data from the client. If the read function is successful (data is available), RC is equal to the data length. A return code of 0 means "end of file". A return code of -1 indicates that an error occurred.

9 Incoming data is interpreted as an item number (in the SocketData data structure). However, if the "item number" is END, the server ends. Otherwise, the corresponding record is read from the ITEMS file.

10If the item number was not found question marks are moved to the item description, and 0 is moved to the unit price.

11Item data is sent to the client by the write function over the SD2 socket. The return code is equal to the data length if the write function was successful or -1 if it was unsuccessful.

12After the program leaves the infinite loop after the END request, sockets are closed and the server program ends.

SSERVER program: Error handling

The error handling subprocedure uses the C language functions: errno and strerror.

```
PErrorHdlr        B

 *   Error handling subprocedure prototype
D ErrorHdlr       Pr
D  DumpText                     12    Value

 *   Error handling subprocedure interface
D ErrorHdlr       PI
D DumpText                      12    Value

C                   Eval      ErrNo@ = GetErrNo             1
C                   Eval      ErrMsg@ = StrError(ErrNo)        2
C     DumpText      Dump
C                   CallP     Close(SD2)
C                   CallP     Close(SD)
C                   Eval      *InLR = *ON
C                   Return

PErrorHdlr        E
```

1 GetErrNo function maps to the C function *errno*. It gets the error number from the system to the ErrNo variable.

2 The StrError function maps to the C function *strerror*. It gets the message text based on the ErrNo variable and places it in the ErrMsg variable.

For our example to run correctly, the SSERVER program must be started first and then the client SCLIENT is called interactively. If you reverse the order and start the client first, the client ends abnormally with a dump because it cannot connect to a socket.

Consider the following points:

- If the server program is running interactively and closes its sockets before it normally ends, the sockets are reused after the server is run again in the same session.
- While the server and the client are running, you may simulate an abnormal termination of the client by using the SysRq key, option 2. The server does not end but waits for data in its read function. If you restart the client later, it never connects because the server wants to *read* data and not *accept* the connect request. You would have to end the server *job* and start the job and both programs again.
- If the server runs in an interactive session and ends abnormally (you can simulate it by using the SysRq key, option 2), the local socket address remains locked until the job ends. When you restart the server later in the same session from the command line, it ends up with the error number 3420, "Address already in use" (ErrNo and ErrMsg variables). Ending the activation group QILE with the RCLACTGRP command does not help. You must end the job (signoff) to unlock the socket address.

  An invocation exit program can be written (using the atiexit function) to close the socket properly after a SysRq 2.

5.5.3.2 The SCLIENT program

To understand the logic of this program better, we break it up into the logical pieces of data definition and procedure.

SCLIENT program: Data definition

Here is the source listing of program SCLIENT:

```
H DFTACTGRP(*NO) ACTGRP('QILE') BNDDIR('QC2LE')

 *   Workstation file to request and display data from the server
FITEMSW    CF   E             WORKSTN

 *   Necessary procedure prototypes with data definitions
 /COPY SCKSRC,SCKCPY

D SocketData    E Ds                  ExtName (ITEMS)
D SockDtaLen      S             10I 0 Inz (%Size(SocketData))
D PortNumber      S             10I 0 Inz(3005)               1
D SD              S             10I 0
D RC              S             10I 0

 *   Server name parameter
D ServerName      S            255A   Inz('LOCALHOST')       2

 *   Error handling subprocedure prototype
D ErrorHdlr       Pr
D DumpText                      12    Value
```

1 Note that the port number is the same as in the SSERVER program data definition in 5.5.3.1, “The SSERVER program” on page 200.

2 For this simple test application, we connect to the same local host on which the server program is also running.

SCLIENT program: Procedure

The client program gets its sockets, obtains an IP address for it, connects to the server and enters a processing loop where data is exchanged.

```
 *   Obtain a socket descriptor
C                   Eval      SD = Socket( AF_INET : SOCK_STREAM : 0)

 *   If socket failed - End the client program with dump
```

```
C                   If        SD < 0
C                   CallP     ErrorHdlr ('ClientSocket')
C                   Return
C                   EndIf

 *   Fill in necessary fields in the IP address structure
C                   Eval      SocketAddr = *ALLX'00'          1
C                   Eval      SinFamily = AF_INET
C                   Eval      SinPort = PortNumber

 *   Prepare the host name for the GetHostByName function      2
C                   Eval      ServerName = %Trim(ServerName) + X'00'
C                   Eval      Server@ = %Addr(ServerName)

 *   Get the host address if given the server name
C                   Eval      Host@ = GetHostByName(Server@)   3

 *   If host name cannot be resolved - End the client program
C                   If        Host@ = *NULL                    4
C                   CallP     ErrorHdlr ('ClientHostN')
C                   Return
C                   EndIf

 *   Set the pointer to the host entry data structure
C                   Eval      HostEntData@ = HName@             5

 *   Copy the IP address from the host entry structure into
 *   the server IP address structure
C                   Eval      SinAddr = HAddrArr(1)             6

 *   Connect to the server
C                   Eval      RC = Connect( SD:                 7
C                                           %Addr(SocketAddr) :
C                                           %Size(SocketAddr) )
 *   If connect unsuccessful - End the client program with dump
C                   If        RC < 0
C                   CallP     ErrorHdlr ('ClientConnect')
C                   Return
C                   EndIf
 *   Write/read loop
C                   DoW       0 = 0

 *   Request the user to enter an item number
C                   ExFmt     ITEMSW0                            8

 *   If F3 pressed - Leave the loop and end the client
C                   If        *In03                              9
C                   Leave
C                   EndIf

 *   Send the item number to the server over the socket
C                   Eval      RC = Write( SD : %Addr(SocketData)
C                                    10      :  SockDtaLen )

 *   If write failed - End the client program with dump
C                   If        RC < 0
C                   CallP     ErrorHdlr ('ClientWrite')
C                   EndIf

 *   If input from the screen was END - End the client program
 *   (which causes the server to end, too)
C                   If        ITEMNBR = 'END'                    11
C                   Leave
C                   EndIf

 *   Read the reply data from the server                         12
C                   Eval      RC = Read ( SD : %Addr(SocketData) :
C                                           SockDtaLen )

 *   If read failed - End the client program with dump
C                   If        RC < 0
C                   CallP     ErrorHdlr ('ClientRead')
C                   Leave
C                   EndIf

 *   If no data available from the server -
 *                  - End the server program with dump
C                   If        RC = 0                             13
```

```
C                   CallP     ErrorHdlr ('ClientRead2')
C                   Leave
C                   EndIf

 *   Display data received from the server
C                   ExFmt     ITEMSW1                            14

 *   If F3 pressed - End the client program
C                   If        *In03
C                   Leave
C                   EndIf

 *   End write/read loop
C                   EndDo

 *   End the program
C                   CallP     Close(SD)
C                   Eval      *InLR=*On
C                   Return
```

1 After obtaining the socket number, SD, the client prepares to get an IP address of the host (server). The SocketAddr data structure is initialized by binary zeroes. The AF_INET number and port number are assigned.

2 – **3** The server name and its address (pointer) **2** are parameters for the GetHostByName function.

4 – **6** If the *GetHostByName* function is not successful **4**, the client program ends. Otherwise, a pointer to an additional data structure containing an array of IP addresses is obtained **5**, and the first IP address from the array is moved in the server address structure **6**.

7 The *connect* function requests the server that the SD socket be connected to the host (by the accept function in the server).

8 – **9** After entering the processing loop, the client shows a display format to get an item number from the keyboard **8**. If the user presses the F3 key, the client ends up normally **9**.

10 The *write* function sends the item number (or END) to the server.

11 If END was sent, the client ends up normally.

12 The *read* function receives the reply from the server, which can be the contents of an item record or question marks (if the server did not find the record).

13 If the return code of the read function is -1, the client program ends with a dump. If the return code is 0, no data was received (end of data) and the client also ends with a dump.

14 Data received from the server is displayed on the workstation. After the Enter key is pressed, the processing loop is repeated. If the F3 key is pressed, the client ends normally.

Try it yourself

The server and client programs are created from the corresponding modules SSERVER and SCLIENT using the following instructions:

```
CRTBNDRPG PGM(RPGISCOOL/SSERVER) SRCFILE(RPGISCOOL/SCKSRC)
CRTBNDRPG PGM(RPGISCOOL/SCLIENT) SRCFILE(RPGISCOOL/SCKSRC)
```

The programs run in the QILE activation group. Prior to compiling the modules, the physical file ITEMS and display file ITEMSW must have been created, as described in 5.5.8, “Running the examples” on page 232. You can use the instructions in the same section to run the program.

5.5.4 Server SSERVER2 and client SCLIENT2 with recovery

In our first example, we did not include any error recovery capability. The second example is more robust. In case one of the programs ends abnormally, the other program does not wait for data in the read operation, but recognizes that the read function failed. The write function is handled the same way, but the chances are much less for a failure.

Figure 26 illustrates the recovery process. Solid ovals represent normal processing without errors. Dotted ovals are loops that are entered in case of a failure.

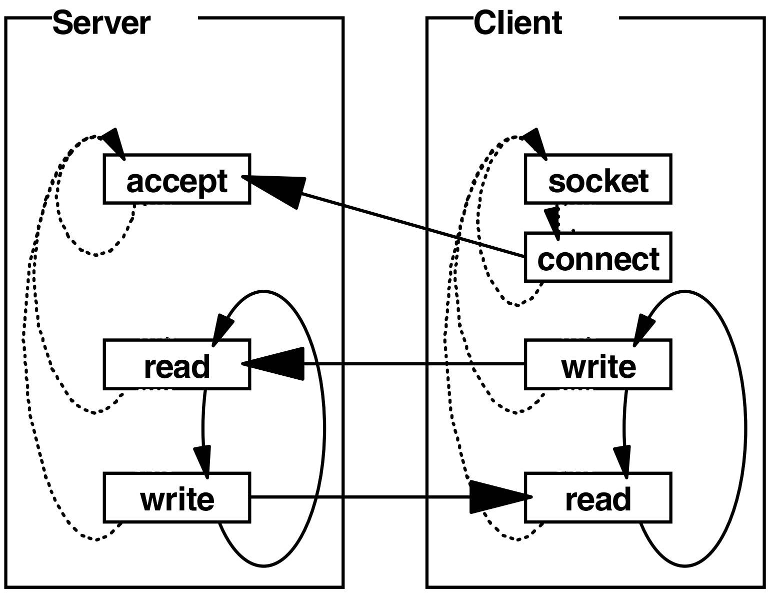


Figure 26. Recovery from failures

The actual recovery process is explained here:

1. In the server program, a failure of the accept, read, or write function is solved by reentering the accept function. The accept function waits for a connect request from the client.
2. In the client program, a failure of the connect, write, or read functions is solved by entering a loop where the socket and connect functions alternate. The socket is closed before the socket function is issued.

3. The server now listens to a maximum of 10 clients and allows up to 10 clients to communicate.
4. If the client's job is abnormally ended, the server's read operation fails. The server recovers from the failure by reissuing an accept function. If the accept function should itself fail, it is repeated until a connect request comes.
5. The server survives if clients end abnormally in their jobs, or even if their jobs end.
6. If a client is started before the server is running, its connect function fails. In this case, the client closes its socket, obtains a new socket, and issues the connect function. This process is repeated until the connect request is accepted by the server, or until the client is ended by the user.
7. More than one client can be started (interactively), but only one can be accepted by the server at a time. The accepted client communicates with the server as long as it wants. After this client ends, the next client is accepted by the server.

The recovery process is performed in the SSERVER2 and SCLIENT2 example programs. These programs are functionally identical to those in 5.5.3, "Example of a simple server SSERVER and client SCLIENT" on page 200, except for the recovery. Only the relevant parts of the programs are shown.

5.5.4.1 The SSERVER2 program with recovery

Here are the relevant parts of the source code for the SSERVER2 program:

```
...

C                   Eval      RC = listen (SD: 10)                  1
...

 *   Recovery loop
C                   DoW       0 = 0

 *   Accept incoming connection request from the client.
 *   A new socket (SD2) is created for the client.
C                   Eval      SD2 = Accept(SD: SockAddr: AddrLen)
C                   If        SD2 < 0

 *   If accept failed - Repeat it with after a delay
C                   CallP     Sleep(1)                               2
C                   Iter
C                   EndIf
...

 *   Read/write loop
C                   DoW       0 = 0

 *   Read data from socket
C                   Eval      RC = Read (SDR: SocketData@: SockDtaLen)

 *   If read failed - Enter recovery loop
C                   If        RC < 0                                  3
C                   Leave
C                   EndIf
...

 *   Write the record to the client
C                   Eval      RC = Write(SDR: SocketData@: SockDtaLen )

 *   If write failed - Enter recovery loop
C                   If        RC <= 0                                  4
C                   Leave
C                   EndIf

 *   End read/write loop
C                   EndDo
```

```
 *   End recovery loop
C                   EndDo
```

1 The program listens at most to 10 clients.

2 If the accept function fails, the program enters a new iteration of the recovery loop after a 1 second delay.

3 – **4** If a read or write function fails, the program leaves the processing loop and enters the recovery loop.

5.5.4.2 The SCLIENT2 program with recovery

Here are the relevant parts of the source code for the SCLIENT2 program:

```
...
 *   Recovery loop (repeats if any failure occurred)
C                   DoW       0 = 0                         1

 *   Obtain a socket descriptor
C                   Eval      SD = Socket(AF_INET: SOCK_STREAM: 0)

...

 *   Connect to the server
C                   Eval      RC = Connect(SD
C                                           : %Addr(SocketAddr)
C                                           : %Size(SocketAddr))

 *   If connect unsuccessful - Repeat connect with delay
C                   If        RC < 0
C                   CallP     Sleep (1)                       2
C                   Iter
C                   EndIf

 *   Write/read loop
C                   DoW       0 = 0

 *   Request the user to enter an item number
C                   ExFmt     ITEMSW0

 *   If F3 pressed - End the client program
C                   If        *In03
C                   ExSr      Terminate
C                   EndIf

 *   Send the item number to the server over the socket
C                   Eval      RC = Write( SD : %Addr(SocketData)
C                                             : SockDtaLen )

 *   If write failed - Enter the recovery loop
C                   If        RC < 0
C                   Leave                                     3
C                   EndIf
...

 *   Read the reply data from the server
C                   Eval      RC = Read (SD: %Addr(SocketData)
C                                           : SockDtaLen )

 *   If read failed - Enter the recovery loop
C                   If        RC <= 0
C                   Leave                                      4
C                   EndIf
...

 *   End read/write loop
C                   EndDo

 *   Close the socket before obtaining a new one
C                   CallP     Close (SD)                        5

 *   End recovery loop
C                   EndDo
```

1 The recovery loop is placed at the beginning of the program. The socket() function must be repeated if the connection with the server has been lost.

2 If the connect() function fails, the program waits one second and then enters the new iteration of the recovery loop.

3 – **4** If the write() or read() function fails, the program leaves the processing loop and enters the recovery loop.

5 The socket is closed at the end of the recovery loop before a new socket is obtained by the socket() function.

Try it yourself

The server and client programs are created from the corresponding modules SSERVER and SCLIENT using the following instructions:

```
CRTBNDRPG PGM(RPGISCOOL/SSERVER2) SRCFILE(RPGISCOOL/SCKSRC)
CRTBNDRPG PGM(RPGISCOOL/SCLIENT2) SRCFILE(RPGISCOOL/SCKSRC)
```

The programs run in the QILE activation group. Prior to compiling the modules, the physical file ITEMS and the display file ITEMSW must have been created, as described in 5.5.8, “Running the examples” on page 232. You can use the instruction in the same section to run the program.

5.5.5 Communication with multiple sockets (multiple I/O)

So far our examples only allow a one to one connection. Only one client talks to a server at a time. In practice, you will often require a method where the server communicates with multiple clients at the same time. The select() function "Wait for events on multiple sockets" can be used to serve this purpose.

5.5.5.1 The select() function (wait for multiple sockets)

The select() function is used to enable an application to multiplex I/O. By using the select function, an application with multiple interactive I/O sources avoids blocking on one I/O stream, while the other stream is ready. For example, an application that receives inputs from two distinct communication endpoints (using sockets) can use the select function to sleep (wait) until input is available from either of the sources. When input is available, the application wakes up and receives an indication as to which descriptor is ready for reading.

The application identifies descriptors to be checked for read, write, and exception status and specifies a time-out value. If any of the specified descriptors is ready for the specified event (read, write, or exception), the select function returns, and indicates which descriptors are ready. Otherwise, the process waits until one of the specified events occur or the wait times out. The indication is in the form of a set of bits representing one socket each. The set is called a *file descriptor set* or *fd_set*, because it has the same format for IFS files and is handled the same way. There are three kinds of file descriptor sets: read_set, write_set, and exception_set.

A descriptor is returned by the select() function in the bit set specified by read_set to indicate one of the following events:

- A read request is pending on a socket descriptor. This occurs when a client is sending data to the server using send() or write() function. In this case, a non-listening socket is returned in the bit set specified by read_set. The server can then issue a recv() or read() function to receive the data.

- A connection request is pending on a socket descriptor. This occurs when a client is connecting using connect() function. In this case a listening socket is returned in the bit set specified by read_set. The server can then issue an accept() function to accept the connection.
- An error event exists on the descriptor. The server should handle the error condition.

Similar rules hold true for write and exception bit sets.

The select() function has the following prototype:

```
 *   int select(int max_descriptor,
 *              fd_set *read_set,
 *              fd_set *write_set,
 *              fd_set *exception_set,
 *              struct timeval *wait_time)

D Select          Pr            10I 0 ExtProc('select')

D  MaxDescr                     10I 0 Value
D  ReadSet                        *   Value
D  WriteSet                       *   Value
D  ExceptSet                      *   Value
D  WaitTime                       *   Value
```

Table 56 shows the parameters for this function.

Table 56. Parameters for the select() function

Argument	Description	Use	RPG data type	C data type
max_descriptor	Maximum number of a descriptor counted from 0	Input	Integer(10)	int
read_set	Set of descriptors tested to be ready for reading	Input	Pointer to a structure	fd_set *
write_set	Set of descriptors tested to be ready for writing	Input	Pointer to a structure	fd_set *
exception_set	Set of descriptors tested for pending exceptions	Input	Pointer to a structure	fd_set *
timeval	Wait time in seconds or microseconds	input	Pointer to a structure	wait_time *

The return values are:

-1	Unsuccessful
0	If the time limit expires
n	The total number of descriptors in all sets that met selection criteria

The *fd_set* structure has the following format in the RPG IV language:

```
D FD_Set          Ds

D  FDes                         10U 0 Dim(7)
```

FDes (File Descriptor) is an array of unsigned integers, each bit of which represents one socket. You must specify at least seven because it is the minimum number required for the set. Such a set represents 224 sockets (32 times 7). Figure 27 on page 212 shows the set.

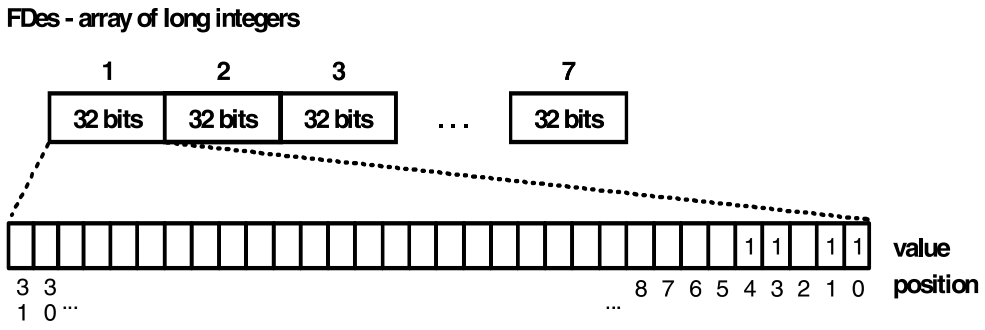


Figure 27. Socket descriptor bits in the array of integers

Only the first integer is illustrated in a detailed view. Four description bits are set to one (1), the others are zeros. Correspondence between bit positions and their decimal and hexadecimal values is shown in Table 57.

Table 57. Socket description bit values

Power of 2	Decimal/Hexadecimal	Description
2^0	1/1	Socket number 0
2^1	2/2	Socket number 1
2^3	8/8	Socket number 3
2^4	16/10	Socket number 4

The sum of the bit values is decimal 27 or hexadecimal X'1B'. You can see such a number in the memory dump or when debugging the program.

To help manipulate the description sets, additional functions are available. They are:

FDZero Removes all descriptors from the set.

FDClr Moves descriptor n from the set.

FDSet Adds descriptor n to the set.

FDIsSet Returns a nonzero value if a descriptor is returned in the set. Otherwise, a zero value is returned.

The functions are realized as subprocedures of the SCKSELF module, which is bound to the SSERVER3 module to create the SSERVER3 program. They use an arithmetic method with powers of two (2) to simulate C-language bit operations.

5.5.5.2 Auxiliary functions to manipulate socket description bits

The FDZero function is used to clear the entire set of description bits before the select function is first used.

```
 *-----------------------------------------------------------------
 *   FDZero - Zero all socket descritpion bits in FDes array
 *            FDes is an array of 7 integers
 *-----------------------------------------------------------------

PFDZero          B                 EXPORT

 *   FDZero subprocedure prototype
D FDZero         Pr
D  FDes                       10U 0 Dim(7)
```

```
 *   FDZero subprocedure interface
D FDZero          PI
D  FDes                         10U 0 Dim(7)

C                   Eval      FDes = 0

PFDZero           E
```

The FDSet function is used to set the description bit representing the socket you want to test for incoming connect() or send() requests before the select() function is issued. You add a bit to your socket description set. The select() function then sets on (to one) only those bits that represent sockets ready to connect. The other bits from your set are set off (to zero).

```
 *-------------------------------------------------------------
 *
 *   FDSet - Set socket description bit
 *
 *   Parameters:
 *   FD    A socket description number
 *   FDes  An array of 7 integers (socket descriptor bits)
 *
 *   Return value: none
 *
 *------------------------------------------------------------

PFDSet            B                   EXPORT

 *   FDSet procedure prototype
D FDSet           Pr
D  FD                           10I 0 Value
D  FDes                         10U 0 Dim(7)

FDSet procedure interface
D FDSet           PI
D  FD                           10I 0 Value
D  FDes                         10U 0 Dim(7)

D Idx             S              5P 0
D FD32            S              5P 0
D RemFD           S              5P 0

 *   Socket number is divided by 32 (number of bits in intege
 *   1 is added because arrays are numbered from 1.

C     FD            Div       32            FD32
C                   MvR                     RemFD
C                   Eval      Idx =  FD32 + 1                    1

 *   A bit is set in the integer which represents the socket.
 *   This is done by adding a power of 2. The exponent is
 *   the remainder modulo 32.

C                   If        FDIsSet(FD: FDes) = 0
C                   Eval      FDes(Idx) = FDes(Idx) + 2 ** RemFD   2
C                   EndIf

PFDSet            E
```

1 The socket number is divided by 32 (number of bits in integer). One (1) is added because arrays are numbered from one.

2 A bit is set in the integer that represents the socket. This is done by *adding* a power of two (2). The exponent is the remainder modulo 32.

The FDClr function clears (sets to zero) a socket description bit. It can be used to remove a socket from your set.

```
 *-------------------------------------------------------------
 *   FDClr - Clear socket description bit
 *
 *   Parameters:
 *   FD    A socket description number
 *   FDes  An array of 7 integers (socket descriptor bits)
 *
 *   Return value:  none
 *
 *-----------------------------------------------------------------
PFDClr            B                   EXPORT

D FDClr           Pr
D  FD                            10I 0 Value
D  FDes                          10U 0 Dim(7)

D FDClr           PI
D  FD                            10I 0 Value
D  FDes                          10U 0 Dim(7)

D Idx             S               5P 0
D FD32            S               5P 0
D RemFD           S               5P 0

 *   Socket number is divided by 32 (number of bits in integer)
 *   1 is added because arrays are numbered from 1.

C     FD            Div       32            FD32                 1
C                   MvR                     RemFD
C                   Eval      Idx = FD32 + 1

 *   A bit is set in the integer which represents the socket.
 *   This is done by subtracting a power of 2. The exponent is
 *   the remainder modulo 32.

C                   If        FDIsSet(FD: FDes) > 0              2
C                   Eval      FDes(Idx) = FDes(Idx) - 2 ** RemFD
C                   EndIf


PFDClr            E
```

1 The socket number is divided by 32 (number of bits in an integer). One (1) is added because arrays are numbered from one.

2 A bit is set in the integer which represents the socket. This is done by *subtracting* a power of two (2). The exponent is the remainder modulo 32.

The FDIsSet function is used after the Select() function completed successfully and returns a set of sockets (description bits) that are ready to connect or send. You test whether your specific socket is contained in this set. If it is contained in the set, you can continue by accepting the connect request or receive data for this particular socket. You accept a connection for a listening socket; you receive data for a non-listening socket.

```
 *-----------------------------------------------------------------
 *   FDIsSet - Test if a socket description bit is set on
 *
 *   Parameters:
 *   FD    A socket description number
 *   FDes  An array of 7 integers (socket descriptor bits)
 *
 *   Returned integer value:
 *   1 - if the bit is set on
 *   0 - if the bit is set off
 *
 *-----------------------------------------------------------------

PFDIsSet          B                   EXPORT

D FDIsSet         Pr             10I 0
D  FD                            10I 0 Value
D  FDes                          10U 0 Dim(7)

D FDIsSet         PI             10I 0
D  FD                            10I 0 Value
D  FDes                          10U 0 Dim(7)

D Idx             S               5P 0
```

```
D FD32            S              5P 0
D RemFD           S              5P 0
D ShiftFD         S              5P 0
D ShiftFD2        S              5P 0
D RemShiftFD2     S              5P 0

 *   Socket number is divided by 32 (number of bits in integer)
 *   1 is added because arrays are numbered from 1.

C     FD            Div       32            FD32
C                   MvR                     RemFD
C                   Eval      Idx = FD32 + 1                       1

 *   Shift the bits right by the socket number (remainder modulo 32)

C                   Eval      ShiftFD = FDes(Idx) * 2 ** -RemFD    2

 *   Return 1 if odd, 0 if even.

C     ShiftFD       Div       2             ShiftFD2
C                   MvR                     RemShiftFD2
C                   Return    RemShiftFd2                          3

PFDIsSet           E
```

1 The socket number is divided by 32 (number of bits in integer). One (1) is added because arrays are numbered from one.

2 Shift the bits right by the socket number (remainder modulo 32). This is done by dividing the socket bit number by a power of two. The exponent is the remainder after dividing the socket number by 32.

3 Return one (1) if the result is odd, or zero (0) if it is even.

5.5.6 Example of multiple I/O

This example shows how the server can use the select function to communicate with multiple clients at the same time. Figure 28 illustrates this configuration. The data exchange is one way only. A client sends an item number, and the server prints out the corresponding item record which it finds in a database file. If, however, the item number is “END”, the client ends and is also disconnected by the server. If the item number is “ENDSV”, the client ends and the server disconnects all clients and ends, too. This example can be extended by an answer message (perhaps an acknowledgement) from the server to the client if you need two-way communication.

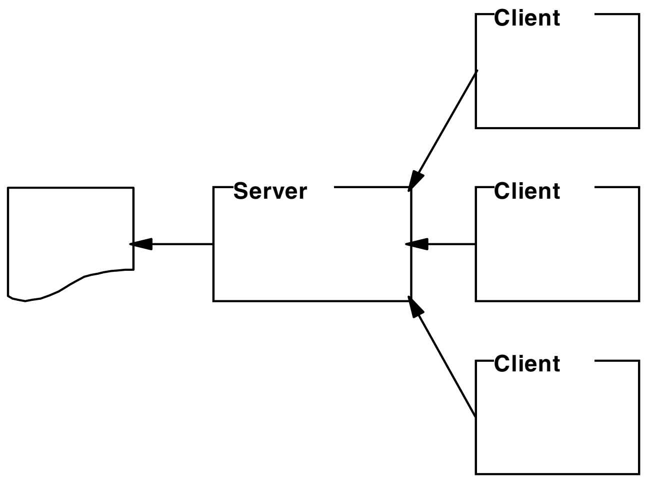


Figure 28. Server accepting multiple clients

Figure 29 is a simplified illustration of how the select(), accept(), and recv() functions are rotated in the server program and how the connect() and send() functions are arranged in the client programs. The connect() is performed only once in the client program. The select() function in the server recognizes both connect() and send() requests.

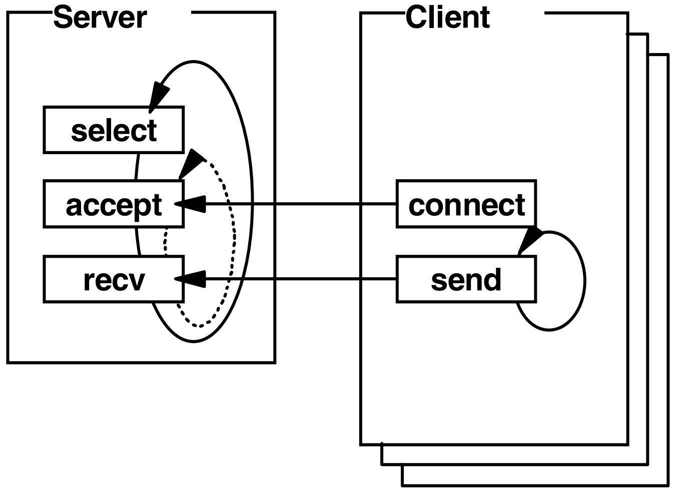


Figure 29. Repeating socket functions in the server and client programs

The main processing loop (solid oval) begins with select() function which recognizes all waiting requests from those sockets we specify in FDes array (readset). The select() function sets corresponding bits in FDes array to one (1) for sockets that actually made a request (connect() or send() function from clients). The other bits are cleared (set to zero).

The server program must find out which bits are set to one (1) after the select() function completed. This is done in the inner loop (dotted oval) where it is decided for each bit if it corresponds to the listening socket or not. If it is the listening socket, the accept() function accepts the connect() request. Otherwise, the recv() function receives data from the send() request.

After the inner loop ends, the original setting of the FDes array is restored so that it tells the select() function which sockets it should test for activity.

5.5.6.1 Server program SSERVER3 communicating with multiple clients

To understand the logic of this program better, we break it up into the logical pieces of the data definition and procedure.

The SSERVER3 program: Data definition

Here is the source code for the data definition portion of the program SSERVER3:

```
 *   Database items file
FITEMS     IF   E           K DISK
FREPORT    O    E             PRINTER OflInd(*In50)             1

 *   Necessary procedrure prototypes with data definitions
 /COPY SCKSRC,SCKCPY

 *   Socket data
D SocketData    E Ds                 ExtName (ITEMS)
 *   Socket data pointer
D SocketData@     S              *   Inz(%Addr(SocketData) )
 *   Socket data  length
D SockDtaLen      S            10I 0 Inz(%Size(SocketData))
```

```
 *   Port number
D PortNumber      S             10I 0 Inz(3005)
 *   Socket description number for the server
D SD              S             10I 0
 *   Socket description number for the client
D SD2             S             10I 0
 *   Current maximum of active socket numbers
D CurMax          S             10I 0                     2
 *   Sockets flags - active/inactive ('1'/'0')
D SckFlags        S              1A   Dim(20)             3
 *   Return code for sockets
D RC              S             10I 0
 *   Option name for SetSockOpt function
D OptVal          S             10U 0 Inz(1)
 *   Pointer to the socket description bit set (readset)
D FD_Set@         S               *   Inz(%Addr(FD_Set))  4
 *   Index variables
D I               S             10P 0
D J               S             10P 0
 *   Wait time in seconds for select function
D WaitTime        Ds
D  Seconds                      10I 0 Inz(300)           5
D  MicroSec                     10I 0 Inz(0)
```

1 The program uses the printer file to print out the item record requested from a client. Since there are multiple clients, records in the printer file are written in the sequence in which the individual clients send data. No feedback information is sent from the server to the clients in this example.

2 We try to keep the set of active sockets possibly minimal. CurMax is the maximal socket number of all sockets that are currently open (but not necessarily sending data). Each time a client socket closes, the current maximum is decremented accordingly.

3 SckFlags is an array of one character elements which can take values “1” or “0”. They reflect current status of each socket: 1 – active, 0 – inactive. We register only 20 sockets, but they can be up to 224 (7 x 32 bits in FDes as defined in the SCKCPY member).

4 Pointer FD_Set@ to the FDSet structure is defined here while FDSet structure itself (containing FDes array) is copied from the SCKCPY copy member.

5 WaitTime structure defines the timeout in seconds and microseconds for the select() function. If no input comes from clients in this time frame, the select() function ends with return code zero (0) and clears the FDes array.

The SSERVER3 program: Procedure

Here is the source code for the main procedure portion of the program SSERVER3:

```
 . . .                                                             6

 *   Clear readset array
C                   CallP     FDZero(FDes)                         7
C                   Eval      SckFlags = '0'

 *   Set on the listening socket in readset and in the flag array
C                   CallP     FDSet(SD: FDes)                      8
C                   Eval      SckFlags(SD + 1) = '1'
C                   Eval      CurMax = SD

 *   Processing loop
C                   DoW       0 = 0                                 9

 *   .Restore readset using the flag array
 *    Flag '1' sets on, flag '0' sets off
C     0             Do        CurMax        J                      10
C                   If        SckFlags(J + 1) = '1'
```

```
C                   CallP     FDSet(J: FDes)
C                   Else
C                   CallP     FDClr(J: FDes)
C                   EndIf
C                   EndDo

 *   .Select the sockets that connect or send data
 *    (deselect other sockets)
C                   Eval      RC = Select(CurMax + 1                      11
                                          : FD_Set@: *NULL: *NULL
C                                         : %Addr(WaitTime) )

 *   .If Select timed out - Restore readset and reenter the loop
 *     (there is no incoming request waiting)
C                   If        RC = 0                                      12
C                   Iter
C                   EndIf

 *   .If Select was unsuccessful - End the server program
 *     (a programming error)
C                   If        RC < 0
C                   Eval      *InLR = *On
C                   Return
C                   EndIf

 *   .Process all incoming requests if Select is successful
C                   If        RC > 0                                      13

 *   ..Inspection loop. Inspect all description bits in readset
 *     up to the current maximum and process those which are set on
C     0             Do        CurMax        I

 *   ...If a bit is set on - Process the incoming request
C                   If        FDIsSet(I: FDes) > 0                        14

 *   ....If the bit represents a listening socket - Accept a Connect
C                   If        I = SD                                      15

 *   .....Try to accept the first client in queue
 *        and create a new client's socket - SD2
C                   Eval      SD2 = Accept(SD: SockAddr: AddrLen)

 *   .....Accept OK - Add the socket to readset
C                   If        SD2 >= 0
C                   CallP     FDSet (SD2: FDes)                           16
C                   Eval      SckFlags(SD2 + 1) = '1'

 *   .....Set a new current maximum of used sockets
C                   If        SD2 > CurMax                                17
C                   Eval      CurMax = SD2
C                   EndIf

 *   ....End of Accept
C                   EndIf

 *   ....Else (if not a listening socket) - Receive an Process data
C                   Else                                                  18

 *   .....Receive data from the current client's socket
C                   Eval      RC = Recv(I: SocketData@
C                                        : SockDtaLen: 0)

 *   .....If Recv failed (error) - End the server program
C                   If        RC < 0
C                   Eval      *InLR = *On
C                   Return
C                   EndIf

 *   .....Recv OK - Process data
C                   ExSr      ProcData                                    19

 *   ...End of Accept or Receive (current socket processing)
C                   EndIf

 *   ..End of Process incoming request
C                   EndIf

 *   ..End of Inspection loop
```

```
C                   EndDo

 *   .End of Process all incoming requests
C                   EndIf

 *   End of Processing loop
C                   EndDo
```

6 An introductory sequence is performed first (socket(), SetSockOpt(), bind() and listen() functions).

7 FDes array (readset) is cleared by the *FDZero* auxiliary function and SckFlags array (socket flags) is set to all character zeros (0).

8 The listening socket SD (obtained in the socket() and activated in the listen() function) is put in FDes and SckFlags. While FDes bits are numbered from zero, the SckFlags elements are numbered from one. CurMax (current maximum socket number) is now equal to the listening socket number SD (usually zero).

9 The main processing loop is entered.

10 At the beginning of the main loop, the readset (FDes array) is restored so that it reflects the socket flags maintained throughout the program. A “1” character in SckFlag becomes a one (1) bit in FDes. A “0” character becomes a zero (0) bit. Auxiliary functions *FDSet* and *FDClr* are used to this purpose. Note that the loop goes from zero (0) to the current maximum socket number CurMax. (the first time it may be from zero to zero).

11 The select() function is performed for the maximum socket number which is one greater than the current maximum. This is because a new client that is not yet included in the current maximum may request connection. The FDSet structure (addressed by the FD_Set@ pointer) contains the FDes array which is set the same as the SckFlags (all open sockets). The set always includes the listening socket as a minimum (so the minimum of CurMax is zero (0)).

12 After the select() function ends, its return code is tested. If it is zero, no requests from clients were outstanding during the wait time in which case the readset is completely cleared. The program goes to a new iteration of the main processing cycle. If the return code is negative, it is a programming error meaning that the select() found out an invalid socket number in the readset. A socket number is invalid if it was not opened (by the socket() function) or if it was closed by the program in the meantime. In this case, the program is ended.

13 If the return code is positive, it represents the number of active sockets recognized by the select() function. This number is not used in this example although it could be used to further optimize the processing cycle by counting the actually processed sockets and comparing the count to this return code. The readset now contains at least one active socket with the one (1) bit in FDes. In the following inner loop (from zero (0) to CurMax through I), the readset is inspected and all active sockets are processed. The *FDIsSet* auxiliary function is used to test whether the socket I is active.

14 If the socket I is active it is processed.

15 If the active socket I is the listening socket SD, the program knows that a connect request is waiting. The program issues the accept() function, which creates a new socket (SD2) and also completes a client's connect() function.

16 If the accept() function was successful (the return code SD2 was positive or zero), the new socket SD2 is projected into the readset and into the socket flags. This ensures that this client will be included in the active set again (its socket may have been closed in the meantime) or as a brand new client (socket).

17 If the accepted socket SD2 is greater than the current maximum, it becomes the current maximum.

18 If the active socket I is not the listening socket, it is processed as a client's socket sending data. The recv() function receives the data and sets a return code. If the return code is negative, it is an error and the program is ended.

19 If the return code is positive or zero, the client's send request is processed. A subroutine is used for this purpose so that the main processing loop is shorter.

The ProcData subroutine is as follows:

```
C      ProcData       BegSr

C                     Select

 *    If item number = ENDSV - Close all sockets and end program
C                     When      ITEMNBR = 'ENDSV'                  20

 *    .Close all sockets (disconnecting them)
C      0              Do        CurMax        J                    21
C                     If        SckFlags(J + 1) = '1'
C                     CallP     Close(J)
C                     EndIf
C                     EndDo

 *    .End the server program
C                     Eval      *InLR = *On
C                     Return

 *    If item number is END or no data received - Remove socket flag
 *    and close the socket
C                     When      ITEMNBR = 'END  ' Or RC = 0        22
C                     Eval      SckFlags(I + 1) = '0'
C                     CallP     Close(I)

 *    .If the closed socket was the current maximum -
 *     - Find the nearest lower active socket
C                     If        I = CurMax  And  I > 0             23
C                     Eval      J = I - 1
C                     DoW       SckFlags(J + 1) = '0'
C                     Eval      J = J - 1
C                     EndDo
 *    .Make the highest active socket current maximum
C                     Eval      CurMax = J
C                     EndIf

 *    Process the received data
C                     Other                                         24
 *    .Read corresponding record from the ITEMS file by key
C      ITEMNBR        Chain     ITEMS

 *    .If record not found - Supply question marks as a response
C                     If        Not %Found
C                     Eval      UNITPR  = 0
C                     Eval      ITEMDESC = *All'?'
C                     EndIf

 *    .Write a record to the REPORT printer file
C                     Write     ITEMDETAIL

C                     EndSl

C                     EndSr
```

20 If the “item number” received from the client is ENDSV the server must clean up and end.

21 The cleanup is done so that all active sockets are closed. The program is then endded.

22 If the “item number” received from the client is END or the return code of the Recv function is zero (0), the socket I is closed and removed from the socket flags (I+1st flag will be set to 0).

23 The socket flags array is optimized if the I-th socket is exactly the CurMax. This one and all 0 sockets under it are discounted. The new current maximum will be the highest socket number that remains open. It must be at least zero (0) (the listening socket), if all clients closed.

24 Regular processing of the received data is performed if a normal item number was received (and the return code from the recv() function was positive).

5.5.6.2 Client program SCLIENT3 to communicate with the server in multiple jobs

To understand the logic of this program better, we break it up into the logical pieces of the data definition and procedure.

The SCLIENT3 program: Data definition

Here is the source code for the data definition portion of the program SCLIENT3:

```
 *   Workstation file to request and display data from the server
FITEMSW    CF   E             WORKSTN

 *   Printer output for tracing socket operations
FQSYSPRT   O    F  120        PRINTER OflInd(*InOA)

 *   Necessary procedure prototypes with data definitions
 /COPY SCKSRC,SCKCPY

 *   Socket data
D SocketData    E Ds                 ExtName (ITEMS)
 *   Socket data pointer
D SocketData@     S               *   Inz(%Addr(SocketData))
 *   Socket data  length
D SockDtaLen      S             10I 0 Inz (%Size(SocketData))
 *   Port number
D PortNumber      S             10I 0 Inz(3005)
 *   Socket description number for the client
D SD              S             10I 0
 *   Return code for sockets
D RC              S             10I 0
 *   Option name for SetSockOpt function
D OptVal          S             10U 0 Inz(1)
 *   Server IP address in dotted form
D ServerAddr      S             15A   Inz('127.0.0.1')                 1
```

1 The host is now identified by the IP address in dotted form instead of a host name (using the gethostbyname() function) as we did in our previous examples. This method is used when the IP address is stable and you do not have to rely on the host table or a domain name server. Searching for the host name may take quite a long time depending on the size and placing of the table. Resolution from the dotted form to the binary form is fast.

The SCLIENT3 program: Procedure

Here is the source code for the main procedure portion of the program SCLIENT3:

```
. . .                                                              1

 *   Connect to the server
C                   Eval      RC = Connect(SD: %Addr(SocketAddr)    2
C                                        : %Size(SocketAddr))

 *   If Connect unsuccessful - End the client program
C                   If        RC < 0
C                   ExSr      Terminate
C                   EndIf

 *   Processing loop
C                   DoW       0 = 0                                  3

 *   Request the user to enter an item number
C                   ExFmt     ITEMSW0                                4

 *   If F3 pressed - End the client program
C                   If        *In03
C                   ExSr      Terminate
C                   EndIf

 *   Send the item number to the server over the socket
C                   Eval      RC = Send(SD: %Addr(SocketData)        5
C                                       : SockDtaLen: 0)

 *   If Send failed - End the client program
C                   If        RC < 0
C                   ExSr      Terminate
C                   EndIf

 *   If "item number" was END - End the client program
 *                        (ENDSV - End also the server program)
C                   If        ITEMNBR = 'END' Or ITEMNBR = 'ENDSV'    6
C                   ExSr      Terminate
C                   EndIf

 *   End of Processing loop
C                   EndDo

 *---------------------------------------------------------------------
 *   Terminate the client program
 *---------------------------------------------------------------------

C     Terminate     BegSr
C                   CallP     Close(SD)
C                   Eval      *InLR = *On
C                   Return
C                   EndSr
```

1 The introductory sequence of socket operations is performed: socket() (obtaining the socket SD), SetSockOpt(), and bind() (which is optional).

2 The connect() function is performed trying to connect to the server (the select() function in the server should recognize it). If the connect() function fails (for example, when the server is not yet started), the socket SD is closed and the client program is terminated.

3 The processing loop is entered.

4 The ITEMSW0 format is written on the display. The user enters data (an item number, or END, or ENDSV) or presses the F3 key. If the F3 key is pressed, the socket SD is closed and the client program is terminated.

5 The send() function sends data entered to the server. If the return code is negative, an error occurred. The socket SD is closed and the client program is terminated.

6 If data entered was END or ENDSV, the socket SD is closed and the client program is terminated.

Try it yourself

Creating the programs:

The server and client programs are created from the corresponding modules SSERVER3 and SCLIENT3, only SSERVER3 binding the module SCKSELF by copy. This is done by the CL programs SSERVER3B and SCLIENT3B.

Program SSERVER3B creates both modules and binds them into the SSERVER3 program:

```
CRTRPGMOD MODULE(SSERVER3) SRCFILE(SCKSRC) SRCMBR(SSERVER3)

CRTRPGMOD MODULE(SCKSELF) SRCFILE(SCKSRC) SRCMBR(SCKSELF)

CRTPGM PGM(SSERVER3) MODULE(SSERVER3 SCKSELF) BNDDIR(QC2LE) ACTGRP(QILE)
```

Program SCLIENT3B creates the SCLIENT3 program directly using the CRTBNDRPG command:

```
CRTBNDRPG  PGM(SCLIENT3) SRCFILE(SCKSRC) +
SRCMBR(SCLIENT3) DFTACTGRP(*NO) BNDDIR(QC2LE)
```

Both programs run in the QILE activation group.

The programs contain one more printer file, QSYSPRT, for tracing individual socket operations. It is set by calling the subprocedure EventLog. You can observe how the communication between the server and several clients proceeds when you run the programs.

Running the programs:

The SSERVER3 program is best submitted as a batch job, the SCLIENT3 program is called from the command line in several sessions after the server has been started. You end a client by sending END item number or pressing the F3 key. If you send the ENDSV item number, the server is ended and all clients discontinued.

5.5.7 Server and client using non-blocking mode

If you do not want the server to wait for client data for a long time, you can relieve it to process other functions in the meantime (for example, exchange short polling and acknowledgement messages between the server and the client). You can use the *non-blocking mode* of operation for this purpose.

Normally, when the read or write functions are temporarily unable to read or write data, they wait until the data arrives. They are said to *block*. This is typical for the read function. The write function does not normally block. Blocking the read function can be interrupted by either of the following events:

- The program or the job is abnormally ended.
- The client’s *job* that is connected to the socket for which the read function is waiting is ended. Ending the client program while the job keeps running is not enough.

The functions that can be used to set the non-blocking mode are:

fcntl	Performs various actions on open descriptors. It can set a mode of input/output operations to a socket.
ioctl	Performs control functions (requests) on a file descriptor.

We use the fcntl function in the example programs because it has a simpler interface.

5.5.7.1 The fcntl() function to set non-blocking I/O mode

The fcntl() function performs various actions on open descriptors. For example, it sets the non-blocking mode. If the fcntl() function sets the socket I/O mode to non-blocking, the read function always returns immediately. If data is available. it is read but the function will not wait. The program can test the result of the operation. There are three possible alternatives:

- Data is available and the program can continue processing.
- Data is not yet available because it is blocked for some reason. The program should repeat the read function one or more times to get the data.
- The read function does not deliver data even after several trials, in which case the program should close the socket description and start processing from the beginning (for example, obtain a new socket).

The prototype for the fcntl() function is as follows:

```
 *   int fcntl(int file_descriptor, int cmd, int argument);

D FCntl           Pr            10I 0 Extproc('fcntl')

D                               10I 0 Value
D F_SETFL                       10I 0 Value
D O_NONBLOCK                    10I 0 Value  Options(*Nopass)
```

Table 58 shows the parameters for this function.

Table 58. Parameters for the Fcntl function

Argument	Description	Use	RPG data type	C data type
file_descriptor	Socket descriptor	Input	Integer(10)	int
cmd	Command to be performed	Input	Integer(10)	int
argument	Optional parameter needed by certain commands	Input	integer(10)	int

The return values are:

-1	Unsuccessful
0	Successful

The parameter values are:

F_SETFL	A command to set the file or socket parameters. The value is actually 7.
O_NONBLOCK	A value (a bit) required to be set for the socket to be in the non-blocked mode. The value is actually 128 decimals (200 octal and 80 hexadecimal).

We use the fcntl function in the following example to show how both the server and client programs can recover from a failure.

5.5.7.2 Server program SSERVER4 using fcntl() to recover from failure

The server program performs the same function as in our first example. It receives an item number from the client, finds the data record, and sends it to the client. If the server ends for some reason (normally or abnormally), it can be restarted so that it accepts all pending requests from the client. If the server receives the item number "END", both the server and client end.

The server program is composed of a main procedure, which calls subprocedures for reading and writing data with recovery. The subprocedures are placed in a separate module SCKRDWR.

The SSERVER4 program: Data definition

Here is the source code for the data definition portion of the program SSERVER4:

```
H DFTACTGRP(*NO)  ACTGRP('QILE')  BNDDIR('QC2LE')

 *   Database items file
FITEMS     IF   E           K DISK

D SocketData    E Ds                  ExtName (ITEMS)
D SocketData@     S               *   Inz(%Addr(SocketData) )
D SockDtaLen      S             10I 0 Inz(%Size(SocketData))
D PortNumber      S             10I 0 Inz(3005)
D SD              S             10I 0
D SD2             S             10I 0
D RC              S             10I 0
D OptVal          S             10U 0 Inz(1)

 *   Necessary procedure prototypes with data definitions
 /COPY SCKSRC,SCKCPY

 *   ReadSocket subprocedure prototype                          1
D ReadSocket      Pr            10I 0
D  SD                           10I 0 Value
D  SockData@                      *   Value
D  SockDtaLen                   10I 0 Value
D  Retry                        10I 0 Value

 *   WriteSocket subprocedure prototype                         2
D WriteSocket     Pr            10I 0
D  SD                           10I 0 Value
D  SockData@                      *   Value
D  SockDtaLen                   10I 0 Value
D  Retry                        10I 0 Value

 *   Error handling subprocedure prototype                       3
D ErrorHdlr       Pr
D DumpText                      12    Value
```

1 The server program calls three subprocedures. The ReadSocket subprocedure reads data from the client's socket and recovers from error messages.

2 The WriteSocket subprocedure writes data to the client's socket and recovers from error messages.

3 ErrorHdlr handles error messages on demand from the main procedure.

The SSERVER4 program: Procedure

Here is the source code for the main procedure portion of the program SSERVER4:

```
 *   Recovery loop
C                   DoW       0 = 0                              1

 *   Obtain a socket descriptor for itself
```

```
C                   Eval      SD = Socket(AF_INET: SOCK_STREAM: 0)

 *   Allow socket description to be reusable
C                   Eval      RC = SetSockOpt(SD: SOL_SOCKET
C                                                 : SO_REUSEADDR
C                                                 : %Addr(OptVal)
C                                                 : %Size(OptVal))

 *   Bind the socket to an IP address
C                   Eval      SocketAddr = *LOVAL
C                   Eval      SinFamily  = AF_INET
C                   Eval      SinPort    = PortNumber
C                   Eval      SinAddr    = INADDR_ANY
C                   Eval      RC = Bind (SD: %ADDR(SocketAddr)
C                                            : %SIZE(SocketAddr))

 *   Listen to 1 client only
C                   Eval      RC = listen (SD: 1)                   2

 *   Accept incoming connection request from the client.
 *   A new socket (SD2) is created for the client.                  3
C                   Eval      SD2 = Accept(SD: SockAddr: AddrLen)

 *   If accept failed - Repeat it after a delay
C                   If        SD2 < 0
C                   CallP     Sleep(1)                               4
C                   CallP     Close(SD2)
C                   CallP     Close(SD )
C                   Iter
C                   EndIf

 *   Allow client socket description to be reusable
C                   Eval      RC = SetSockOpt(SD2: SOL_SOCKET
C                                                 : SO_REUSEADDR
C                                                 : %Addr(OptVal)
C                                                 : %Size(OptVal))

 *   Set nonblocked mode for the socket                             5
C                   Eval      RC = FCntl(SD2: F_SETFL: O_NONBLOCK)

 *   Read/write loop
C                   DoW       0 = 0                                 6

 *   Read data from socket                                          7
C                   Eval      RC = ReadSocket(SD2: SocketData@
C                                               : SockDtaLen: 5)

 *   If read failed - Enter recovery loop
C                   If        RC <= 0
C                   CallP     Close(SD2)                            8
C                   CallP     Close(SD)
C                   Leave
C                   EndIf

 *   If item number is END - End the server normally
C                   If        ITEMNBR = 'END'
C                   CallP     Close (SD2)
C                   CallP     Close (SD)
C                   Eval      *InLR = *On
C                   Return
C                   EndIf

 *   Read the corresponding record from the parts file by key
C     ITEMNBR       Chain     ITEMS
C                   If        Not %Found

 *   If record not found send question marks to the client
C                   Eval      UNITPR  = 0
C                   Eval      ITEMDESC = *All'?'
C                   EndIf

 *   Write the record to the client
C                   Eval      RC = WriteSocket                        9
C                                   (SD2: SocketData@: SockDtaLen: 5)

 *   If write failed - Enter recovery loop
C                   If        RC < 0                                  10
C                   CallP     Close(SD2)
```

```
C                   CallP     Close(SD)
C                   Leave
C                   EndIf

 *   End read/write loop
C                   EndDo

 *   End recovery loop
C                   EndDo                                   11
```

1 The server enters the recovery loop from the beginning. In case of a failure, the whole procedure of obtaining a new socket, binding it etc., is repeated.

2 The server listens to one client only.

3 – **4** The accept function creates a client's socket **3**. In case of a failure, both sockets are closed, and after a one second delay, the recovery loop is entered **4**. The sleep function is very simple. Its prototype can be found in the SCKCPY /COPY member.

5 After making the new socket description reusable, the fcntl function sets the non-blocking mode to the SD2 socket.

6 The read/write loop is entered and processes the server transactions with the client. This loop is repeated until an error in the ReadSocket or WriteSocket functions occurs or the user ends the processing from the client (by sending an END item number).

7 The ReadSocket function is performed. It has the same parameters as the read function with the last parameter specifying that it should retry five times in case of error.

8 If the ReadSocket function fails, both sockets are closed and the recovery loop is reentered.

9 – **10** The WriteSocket function is performed. The last parameter specifies that it should retry five time in case of error **9**. If the WriteSocket function fails **10**, both sockets are closed and the recovery loop is re-entered.

11 The last statement of the main procedure is the end of the recovery loop.

The SSERVER4 program: ReadSocket and WriteSocket subprocedures
Two subprocedures, ReadSocket and WriteSocket, are identical except for the read or write functions. They serve several purposes:

- Because the read() (or write) function does not wait for data but returns immediately, a test is made for the error code if it is EWOULDBLOCK, regardless of the value in the return code. This code means that data is not yet available but it will be available later. If it was not for the fcntl() function that set the non-blocking mode, the read() function would have waited for the data.
- If the error code is EWOULDBLOCK, the function is repeated after some delay as long as the error code does not change.
- When the error code changes, the inner loop exits and the outer loop is entered. The loop is repeated a predefined number of times (five in our example). If no data is read after five loops, -1 is returned to the caller. Otherwise, the length of data read (written) is returned.
- For some reason, if the program or the procedure fails, the error number and text are available in the corresponding global variables (defined in the SCKCPY /COPY member).

The error code is made available by the GetErrNo() function, which is a remapped __error() C function. The GetErrNo() function gets access to the ErrNo variable through a pointer. The ErrNo variable contains the error code. Before issuing any function that returns an error code, the ErrNo variable should be cleared. However, it is possible only after the GetErrNo() function was performed. If the error code is not cleared before the function is entered, the resulting error code may be false.

```
 *-----------------------------------------------------------------
 *   ReadSocket - Read socket with recovery
 *-----------------------------------------------------------------

PReadSocket       B

 *   ReadSocket subprocedure prototype
D ReadSocket      Pr           10I 0
D  SD                          10I 0 Value
D  SockData@                     *   Value
D  SockDataLen                 10I 0 Value
D  Retry                       10I 0 Value

 *   ReadSocket subprocedure interface
D ReadSocket      PI           10I 0
D  SD                          10I 0 Value
D  SockData@                     *   Value
D  SockDataLen                 10I 0 Value
D  Retry                       10I 0 Value

D RC              S            10I 0

C                   Do        Retry                                 1

C                   Eval      ErrNo@ = GetErrNo                     2
C                   Eval      ErrNo = 0                             3

C                   Eval      RC = Read (SD: SockData@: SockDataLen)
C                   Eval      ErrNo@ = GetErrNo                     4
C                   Eval      ErrMsg@ = StrError(ErrNo)

C                   DoW       ErrNo = EWOULDBLOCK                   5

C                   CallP     Sleep (1)                             6

C                   Eval      RC = Read (SD: SockData@: SockDataLen)
C                   Eval      ErrNo@ = GetErrNo                     7
C                   Eval      ErrMsg@ = StrError(ErrNo)

C                   EndDo

C                   If        RC > 0
C                   Return    RC                                    8
C                   EndIf

C                   EndDo

C                   Return    RC                                     9

PReadSocket       E
```

1 The procedure enters the retry loop that runs at most *Retry* times. Retry is a parameter that specifies the maximum number of iterations.

2 – 3 Before the read function is issued the ErrNo variable should be cleared 3. It is possible only after the GetErrNo function was performed. The GetErrNo function gets access to the ErrNo variable through a pointer 2.

4 The GetErrNo function is performed again after the read function ended. The read function ends immediately because it runs in non-blocking mode. ErrNo variable now contains an error code. Error code 0 means that the operation was successful. Other codes are errors. The GetErrMsg function

is optional. It is good to see the text of the error message in a dump or when debugging the program.

5 The error code is tested if it is EWOULDBLOCK, which is actually the number 3406. If so, it means that the read function would block if running in blocking mode (as if the fcntl function were not performed). In this case, a loop is entered and is repeated as long as the error code is EWOULDBLOCK.

6 A one second delay is performed so that the read function cannot repeat too fast.

7 The read function is issued again and the GetErrNo function immediately after. It is to get a new value of the error number.

8 When the error number changes, the inner loop is left and a test is made if the return code is positive (success in reading data). If positive, the return code value is returned to the caller and the subprocedure ends.

9 If the return code was negative or zero, the outer loop is reentered until the number of retries is exhausted. After that, if the return code is still negative or zero, the subprocedure returns to the caller with this code.

Note that if the data to be read is immediately available, no loop is completed, the first read function reads the data and the procedure returns with the positive return code **6** immediately.

The WriteSocket procedure is literally the same as the ReadSocket procedure, except that it uses the write function.

5.5.7.3 Client program using fcntl() to recover from a failure

The client program performs the same function as in our first example. It sends an item number to the server and receives the data record, which it displays on the screen. This time the client can be started before the server and wait. If the client ends for some reason (normally or abnormally), it can be restarted. If the client sends the "END" item number, both the client and the server end.

The client program is composed of a main procedure which calls subprocedures for reading and writing data with recovery. The subprocedures are placed in a separate module SCKRDWR.

The SCLIENT4 program: Data definition

Here is the source code for the data definition portion of the program SCLIENT4:

```
H DFTACTGRP(*NO) ACTGRP('QILE') BNDDIR('QC2LE')

 *   Workstation file to request and display data from the server
FITEMSW    CF   E             WORKSTN

 *   Necessary procedure prototypes with data definitions
 /COPY SCKSRC,SCKCPY

 *   ReadSocket subprocedure prototype
D ReadSocket      Pr            10I 0
D  SD                           10I 0 Value
D  SockData@                      *   Value
D  SockDtaLen                   10I 0 Value
D  Retry                        10I 0 Value

 *   WriteSocket subprocedure prototype
D WriteSocket     Pr            10I 0
D  SD                           10I 0 Value
D  SockData@                      *   Value
D  SockDtaLen                   10I 0 Value
```

```
D  Retry                        10I 0 Value

D SocketData    E Ds                  ExtName (ITEMS)
D SockDtaLen      S             10I 0 Inz (%Size(SocketData))
D PortNumber      S             10I 0 Inz(3005)
D SD              S             10I 0
D RC              S             10I 0

D ServerAddr      S             15A   Inz('127.0.0.1')

 *   Error handling subprocedure prototype
D ErrorHdlr       Pr
D DumpText                      12    Value
```

The SCLIENT4 program: Main procedure

Here is the source code for the main procedure portion of the program SCLIENT4:

```
 *   Recovery loop
C                   DoW       0 = 0                                 1

 *   Obtain a socket descriptor
C                   Eval      SD = Socket(AF_INET: SOCK_STREAM: 0)  2

 *   Allow socket description to be reusable                        3
C                   Eval      RC = SetSockOpt(SD: SOL_SOCKET
C                                             : SO_REUSEADDR
C                                             : %Addr(OptVal)
C                                             : %Size(OptVal))

 *   Fill in necessary fields in the IP address structure
C                   Eval      SocketAddr = *ALLX'00'
C                   Eval      SinFamily = AF_INET
C                   Eval      SinPort = PortNumber

 *   Copy the IP address from the host entry structure into
 *   the server IP address structure
C                   Eval      SinAddr = InetAddr(%Addr(ServerAddr)) 4

 *   Connect to the server
C                   Eval      RC = Connect( SD:                     5
C                                         %Addr(SocketAddr):
C                                         %Size(SocketAddr))

 *   If connect unsuccessful - Enter recovery loop
C                   If        RC < 0
C                   CallP     Close(SD)                             6
C                   CallP     Sleep(1)
C                   Iter
C                   EndIf

 *   Set nonblocked mode for the socket
C                   Eval      RC = FCntl(SD: F_SETFL: O_NONBLOCK)   7

 *   Write/read loop
C                   DoW       0 = 0                                 8

 *   Request the user to enter an item number
C                   ExFmt     ITEMSW0

 *   If F3 pressed - Leave the loop and end the client
C                   If        *In03
C                   CallP     Close(SD)
C                   Eval      *InLR = *On
C                   Return
C                   EndIf

 *   Send the item number to the server over the socket
C                   Eval      RC = WriteSocket(SD: %Addr(SocketData)
C                                              : SockDtaLen: 5)

 *   If write failed - Enter recovery loop
C                   If        RC < 0
C                   Callp     Close(SD)                             9
C                   Leave
C                   EndIf
```

```
 *   If input from the screen was END - End the client program
C                   If        ITEMNBR = 'END'
C                   Callp     Close(SD)
C                   Eval      *InLR = *On
C                   Return
C                   EndIf

 *   Read the reply data from the server
C                   Eval      RC = ReadSocket(SD: %Addr(SocketData)
C                                                  : SockDtaLen: 5)

 *   If read failed - End the client program with dump
C                   If        RC <= 0
C                   Callp     Close(SD)                                    10
C                   Leave
C                   EndIf

 *   Display data received from the server
C                   ExFmt     ITEMSW1

 *   If F3 pressed - End the client program
C                   If        *In03
C                   Leave
C                   EndIf

 *   End write/read loop
C                   EndDo

 *   End recovery loop
C                   EndDo                                                  11
```

1 The client enters the recovery loop from the beginning.

2 – **5** The *Socket* **2**, *setsockopt* **3**, *inetaddr* **4**, and *connect* **5** functions are performed to connect a new socket to the server. The setsockopt function ensures that the same SD number will be used for socket descriptor.

6 The connect function will fail if the server is not running. In this case, after a one second delay, the socket is closed, and the recovery loop is reentered.

7 The *fcntl* function sets non-blocking mode for the SD socket. This mode holds until the socket is closed.

8 The write/read loop is entered. It is repeated until a read or write error occurs or the user ends the program (by sending the END item or pressing the F3 key).

9 – **10** If a write **9** or read **10** error occurs (for example, the server ended), the socket is closed and the recovery loop is entered.

11 The last statement in the main procedure is the end of the recovery loop.

Try it yourself

The server and client programs are created from the corresponding modules SSERVER4 and SCLIENT4 binding the module SCKRDRW by copy. This is done by the CL programs SSERVER4B and SCLIENT4B.

Program SSERVER4B creates both modules and binds them into the SSERVER4 program:

```
CRTRPGMOD MODULE(SSERVER4) SRCFILE(SCKSRC) SRCMBR(SSERVER4)

CRTRPGMOD  MODULE(SCKRDRW) SRCFILE(SCKSRC) SRCMBR(SCKRDRW)

CRTPGM     PGM(SSERVER4) MODULE(SSERVER4 SCKRDRW)
             BNDDIR(QC2LE) ACTGRP(QILE)
```

Program SCLIENT4B creates both modules and binds them into the SCLIENT4 program:

```
CRTRPGMOD MODULE(SCLIENT4) SRCFILE(SCKSRC) SRCMBR(SCLIENT4)

CRTRPGMOD  MODULE(SCKRDRW) SRCFILE(SCKSRC) SRCMBR(SCKRDRW)

CRTPGM     PGM(SCLIENT4) MODULE(SCLIENT4 SCKRDRW)
             BNDDIR(QC2LE) ACTGRP(QILE)
```

The programs run in the QILE activation group.

5.5.8 Running the examples

You can try running the examples directly or by compiling the source code on your AS/400 system, provided that you have the RPGISCOOL library installed.

You can use the following commands to execute the programs SSERVER and SCLIENT:

```
CALL PGM(RPGISCOOL/SSERVER) interactively in one session or
SBMJOB CMD(CALL PGM(RPGISCOOL/SSERVER)) JOB(SSERVER) in batch
and
CALL PGM(RPGISCOOL/SCLIENT) in another session
```

You can use the following commands to execute the programs SSERVER2 and SCLIENT2:

```
CALL PGM(RPGISCOOL/SSERVER2) interactively in one session or
SBMJOB CMD(CALL PGM(RPGISCOOL/SSERVER2)) JOB(SSERVER2) in batch
and
CALL PGM(RPGISCOOL/SCLIENT2) in another session
```

You can use similar commands to execute the programs SSERVER3, SCLIENT3, and SSERVER4, SCLIENT4. Note that SCLIENT2 and SCLIENT3 can be called in more than one session to test multiple clients.

All examples enable the ending of the client when you press the F3 key or send the item number "END" to the server. Then, the server ends whether it is running in batch or interactive mode. The client that sent the "END" item number ends, too. the other clients (if any) may wait in the EXFMT operation. If you want, you can create the programs by using the following commands:

```
CRTBNDRPG PGM(RPGISCOOL/SSERVER) SRCFILE(RPGISCOOL/SCKSRC) SRCMBR(SSERVER)
CRTBNDRPG PGM(RPGISCOOL/SCLIENT) SRCFILE(RPGISCOOL/SCKSRC) SRCMBR(SCLIENT)
```

You can use similar commands for the SSERVER2, SCLIENT2 couple.

For creation of the programs SSERVER3, SCLIENT3 and SSERVER4, SCLIENT4, there are CL programs SSERVER3B, SCLIENT3B and SSERVER4B, SCLIENT4B that create modules and bind them into the programs.

You can compile the files used in the examples: the ITEMS database file, ITEMSW display file, and REPORT printer file. Their source members are contained in the SCKSRC source file in the RPGISCOOL library and can be found in the following sections.

5.5.8.1 Physical file ITEMS definition and contents

Here is the definition of the physical file ITEMS:

```
 *****************************************************************
 *   ITEMS from SCKSRC in RPGISCOOL
 *
 *   Item master file
 *****************************************************************
A                                      UNIQUE
A          R ITEMSR
 *   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 physical file object can be created with the following command:

```
CRTPF FILE(RPGISCOOL/ITEMS) SRCFILE(RPGISCOOL/CSKSRC)
```

You can use the following file contents for re-creating our example:

Item number	Unit price	Item description
00001	26.78	First item
00002	53.59	Second item
00003	80.38	Third item

5.5.8.2 Display file ITEMSW

Here is the definition of the display file ITEMSW:

```
 ****************************************************************
 *   ITEMSW from SCKSRC in RPGISCOOL
 *
 *   ITEMSW - Workstation file to access ITEMS DB file
 ****************************************************************
A                                      DSPSIZ(24 80 *DS3)
A                                      REF(RPGISCOOL/ITEMS)
A                                      CA03(03 'End')
 *   Format to enter item number
A          R ITEMSW0
A                                  3  2'Enter an item number and press Ent-
A                                      er.'
A                                      DSPATR(HI)
A                                  5  2'Item number....:'
A            ITEMNBR   R        B  5 20
```

```
A                                23  2'F3=Exit'
A                                     COLOR(BLU)
 *   Format to show item record data
A          R ITEMSW1
A                                 5  2'Item number....:'
A                                     DSPATR(HI)
A            ITEMNBR   R        O  5 20
A                                 6  2'Unit price.....:'
A                                     DSPATR(HI)
A            UNITPR    R        O  6 20EDTCDE(K)
A                                 7  2'Item description:'
A                                     DSPATR(HI)
A            ITEMDESC  R        O  7 20
A                                23  2'F3=Exit'
A                                     COLOR(BLU)
A  80                            24  2'Server does not response'
A                                     DSPATR(HI)
```

The Display file object can be created using the following command:

```
CRTDSPF FILE(RPGISCOOL/ITEMSW) SRCFILE(RPGISCOOL/CSKSRC)
```

5.5.8.3 Printer file REPORT

Here is the definition of the printer file REPORT:

```
 *****************************************************************
 *   Member REPORT from ILESRC in RPGISCOOL
 *
 *   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
```

This printer file object can be created with the following command:

```
CRTPRTF FILE(RPGISCOOL/REPORT) SRCFILE(RPGISCOOL/CSKSRC)
```

5.5.9 More information about sockets programming in RPG IV

The essential information on sockets is contained in the following manuals:

- *OS/400 Sockets Programming V4R4*, SC41-5422
- *OS/400 UNIX-Type APIs V4R4*, SC41-5875

These manuals assume that the socket functions are written in the C language, but more specifically in the ILE C/400 language. No manuals are available so far for RPG IV programmers that want to use sockets in their programs until this one.

The C prototypes and named constants used in the socket functions can be found in the QSYSINC library, H source file.

On the other end of the application programming spectrum is Net.Commerce, which enables the sophisticated programmer to do e-business. The redbook *Net.Commerce V3.2 for AS/400: A Case Study for Doing Business in the New Millennium*, SG24-5198, will help you here.

5.6 Writing CGI programs using RPG IV

An HTTP server responds to a browser's request by sending static or dynamic documents. Often there is a need to allow the readers of an HTML document to return information back to the server, or allow the readers to retrieve dynamically certain kinds of information from the server. This request can be realized by using special programs that cooperate with the server.

The mechanism that defines the rules for communication between HTTP server and such programs is called a Common Gateway Interface (CGI). A programmer creates a gateway program and stores it in a library. A user of a Web page calls the gateway program using the link from the Web page. The HTTP server starts the gateway program, which executes and returns the results to the user.

The basic steps for calling a CGI programs are illustrated in Figure 30 and described in the list that follows.

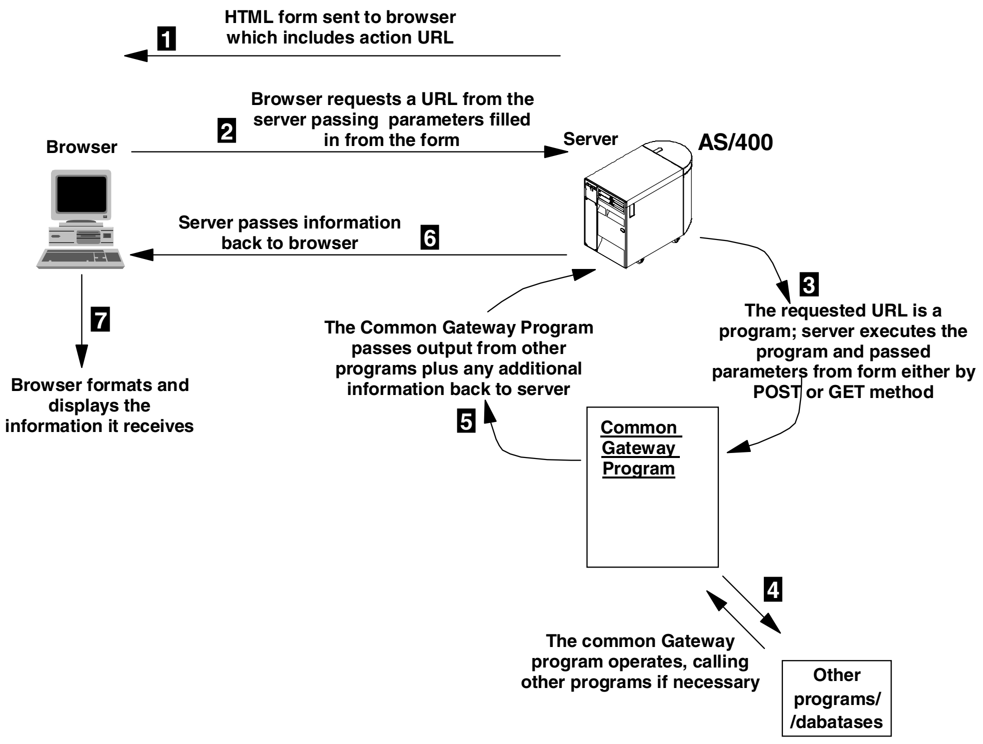

1
HTML form sent to browser which includes action URL
2
Browser requests a URL from the server passing parameters filled in from the form
Browser
Server
AS/400
Server passes information back to browser
6
7
Browser formats and displays the information it receives
The Common Gateway Program passes output from other programs plus any additional information back to server
5
3
The requested URL is a program; server executes the program and passed parameters from form either by POST or GET method
Common Gateway Program
4
The common Gateway program operates, calling other programs if necessary
Other programs/ /dabatases

Figure 30. Basic steps when using the CGI program

1 The server sends an HTML form to the Web client. This form contains an action URL that points to a CGI program located somewhere on the network.

2 The browser sends a request to the server. This request contains a Uniform Resource Locator (URL) with the path and name of the CGI program, along with all input data, entered by the user.

3 The server looks for the CGI program and passes all input data to it. This can be done in two different ways depending on the method (GET or POST) used by the form.

4 The server executes the CGI program, which can call other programs and access a database file without restrictions.

5 As result of its execution, the CGI program creates output in the form of an HTML document and sends it to the server. This is done by writing to standard output (STDOUT).

6 The server forwards the received information to the browser. This is usually in the form of an HTML document, but can be also redirected to another HTML document.

7 Finally, the browser receives this information and displays it to the user.

To do all of this, a set of standards has been set to define the methods for passing input data from the HTTP server to the CGI program, and how output data from the CGI program must be formatted to be accepted by the server.

The URL

The URL specifies the name of the server on the Internet, and location of requested resources, such as a CGI program. It looks similar to this example:

```
protocol://server_name:port/path_name/(?parameters)
```

5.6.1 HTML form document

The FORM tag in the HTML document is one of the more popular ways of prompting for information and identifying the CGI application that will handle the user input. An HTML document with a FORM tag is shown in the following example:

```
<html>
<head>
<title>Registration Form</title>
</head>
<body bgcolor="#F8F8FF">
<h1>Ordering an IBM System AS/400</h1>
<p><b>Please fill in the following: </b>
<form method="POST" action="/cgibin/cgiprog">                           1
<b>Name:</b><br>
<input type="TEXT" name="NAME" size=30 maxlength="40" clear="ALL">      2
<p>
<b>City:</b><br>
<input type="TEXT" name="CITY" size=30 maxlength="40" clear="ALL">      2
<p>
<b>Street:</b><br>
<input type="TEXT" name="STREET" size=30 maxlength="40" clear="ALL">    2
<p>
<b> Which AS/400 would you like to order? </b>
<br>
<input type="RADIO" name="TYPE" value="P1" checked>Portable             2
<input type="RADIO" name="TYPE" value="P2">Server
<input type="RADIO" name="TYPE" value="P3">System
<p>
<b> Do you want the Support Line Service? </b>
<br>
<input type="RADIO" name="SERV" value="T" checked>Yes                   2
<input type="RADIO" name="SERV" value="F">No
<p>
Please order:
<p>
<input type="SUBMIT" value="Order">                                     3
<input type="RESET" value="Clear Form">
</form>
<p>
</body>
</html>
```

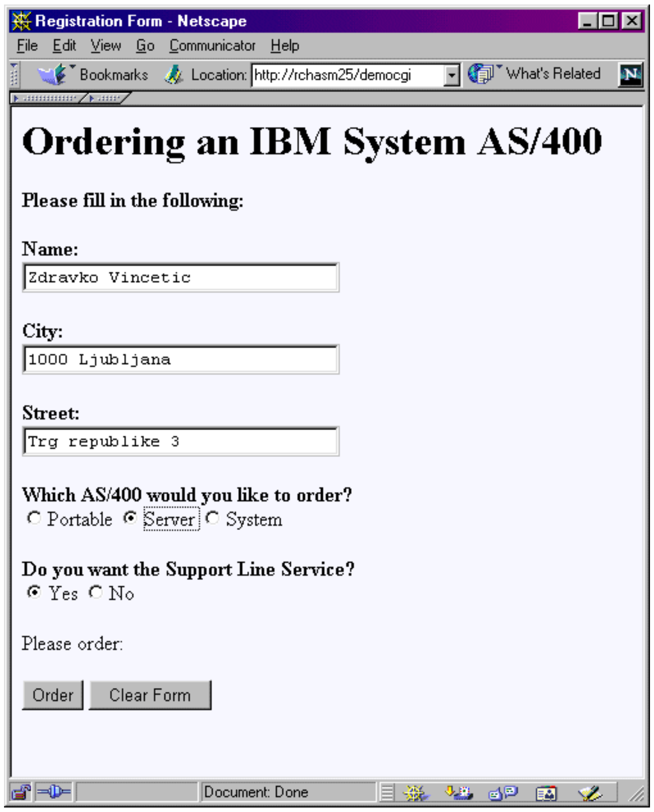
A Form document basically consists of three elements:

1 The Form tag's method can be either GET or POST. The ACTION keyword defines the name of a CGI program to be called and the logical path to this program.

2 The Input tags define the input capable fields on the form document. THe text fields NAME, CITY, and STREET and the radio buttons TYPE and SERV are used to transfer input data entered by the user to the CGI program.

3 The Submit and Reset buttons are used to submit the order or to clear data already entered into the Form document.

When the Web browser displays this HTML Form document, it looks similar to the example Figure 31. We filled in the fields as you would complete the form.


Registration Form - Netscape
File Edit View Go Communicator Help
Bookmarks Location: http://rchasm25/democgi
What's Related

Ordering an IBM System AS/400

Please fill in the following:

Name:
Zdravko Vincetic

City:
1000 Ljubljana

Street:
Trg republike 3

Which AS/400 would you like to order?
Portable [✓ Server] System

Do you want the Support Line Service?
[✓ Yes] No

Please order:

Order
Clear Form
Document: Done

Figure 31. HTML form document displayed by the Web browser

After pressing the button on the Form document to submit it, the input data from the Form document is sent to the CGI program identified by the action keyword. To find this program, the HTTP server looks in the HTTP configuration file for the following directives:

Note

The Pass directive shown is not a good example because it would work only for the one page. This pass statement, would allow all members in the CGISRC file to be passed to the browser:

```
Pass  /democgi   /QSYS.LIB/RPGISCOOL.LIB/CGISRC.FILE/*
```

Also note that any time you make changes to the HTTP server configuration file, you must restart the server for the changes take effect.

```
Pass  /democgi   /QSYS.LIB/RPGISCOOL.LIB/CGISRC.FILE/CGIHTML.MBR
Exec  /cgibin/*  /QSYS.LIB/RPGISCOOL.LIB/*.PGM
```

The Pass directive maps the name democgi with the member CGIHTML in the source file CGISRC in the library RPGISCOOL. This member contains the Form document described earlier. The directive allows access and display of the Form document when a user enters the following URL:

```
http://server-name/democgi
```

The Exec directive maps the logical path /cgibin to the real library RPGISCOOL.LIB, where the CGI program is stored. On the AS/400 system, CGI programs must be stored in the QSYS.LIB file system.

There are two methods that you can use to access data from HTML forms. They use different techniques to encode input data. The two methods are:

GET	The CGI program receives input data in the environment variable QUERY_STRING.
POST	The CGI program receives input data from Standard Input (stdin).

POST is the method recommended for sending data to the CGI programs. The reason is that the length of the environment variable QUERY_STRING, which is used by the GET method, can be limited in length on some server platforms. In some cases, this can cause a loss of data. In the POST method, all data is passed through stdin, and there is no length limitation.

The main advantage of using the GET method is that you can access the CGI program using a form.

For more information on this topic, refer to the redbook *Cool Title About the AS/400 and Internet*, SG24-4815.

5.6.2 Introduction to a service program to aid CGI programing

To write CGI programs with RPG IV, we need support to access environment variables like QUERY_STRING, read from a standard input stream, and write to a standard output stream. This support is available through several APIs included in the service programs.

The old version of the service program, available before V4R3, is QTMHCGI in the library QTCP. The new version of this service program, available from V4R3, is QZHBCGI in the library QHTTPSVR.

The service program QZHBCGI in library QHTTPSVR contains the following seven APIs:

- Get Environment Variable (QtmhGetEnv) API
- Put Environment Variable (QtmhPutEnv) API
- Read from Stdin (QtmhRdStin) API
- Write to Stdout (QtmhWrStout) API
- Convert to DB (QtmhCvtDB) API
- Parse QUERY_STRING Environment Variable or Post stdin data (QzhbCgiParse) API
- Produce Full HTTP Response (QzhbCgiUtils) API

The old service program QTMHCGI, in library QTCP, contains only the first five APIs. There is no difference in behavior or parameters between these APIs in either of the two service programs.

APIs names

Be careful when using the API names because the names of all these APIs are case sensitive.

A detailed description of these APIs are available in the IBM manual *HTTP Server for AS/400 Web Programming Guide*, GC41-5435.

5.6.2.1 Get environment variable (QtmhGetEnv) API

The QtmhGetEnv API allows you to get the value set by the server for a particular HTTP environment variable. The required parameters are shown in the Table 59.

Table 59. Parameters for the API QtmhGetEnv

Num	Description	Use	Data type
1	Receiver variable	Output	Char(*)
2	Length of receiver variable	Input	Integer(10)
3	Length of response	Output	Integer(10)
4	Request variable	Input	Char(*)
5	Length of request variable	Input	Integer(10)
6	Error code	I/O	Char(*)

Here are some of the environment variables supported by the system:

CONTENT_LENGTH	When the POST method is used to send information, this variable contains the number of characters. The CGI program must read CONTENT_LENGTH to determine the length of the data being read from the standard when processing the POST request.
QUERY_STRING	When information is sent by using a GET method, this variable contains the information in a query that follows the "?" (question mark). The string is coded in the standard URL format of changing spaces to "+" (plus signs) and encoding special characters with "%xx" hexadecimal encoding.

REMOTE_ADDR	Contains the requester's IP address.
REMOTE_HOST	Contains the requester's host name.
REQUEST_METHOD	Contains the method (GET or POST) specified with the METHOD attribute in an HTML form used to send the request.

5.6.2.2 Put environment variable (QtmhPutEnv) API

The QtmhPutEnv API allows you to set or create a job-level environment variable. This is useful for communication between programs running in the same job, such as your program and the Net.Data language environment variable DTW_SYSTEM. The required parameters are shown in the Table 60.

Table 60. Parameters for the API QtmhPutEnv

Num	Description	Use	Data type
1	Environment string	Input	Char(*)
2	Length of environment string	Input	Integer(10)
3	Error code	I/O	Char(*)

5.6.2.3 Read from stdin (QtmhRdStin) API

The QtmhRdStin API allows CGI programs written in languages other than C to be read from stdin. CGI programs read from stdin when the request from the browser indicates the method that is POST. This API reads what the server has generated as input for the CGI program. The required parameters are shown in Table 61.

Table 61. Parameters for the API QtmhRdStin

Num	Description	Use	Data type
1	Receiver variable	Output	Char(*)
2	Length of receiver variable	Input	Integer(10)
3	Length of response available	Output	Integer(10)
4	Error code	I/O	Char(*)

5.6.2.4 Write to stdout (QtmhWrStout) API

The QtmhWrStout API provides the ability for CGI programs written in languages other than C to write to stdout. The required parameters are shown in Table 62.

Table 62. Parameters for the API QtmhWrStout

Num	Description	Use	Data type
1	Data variable	Input	Char(*)
2	Length of data variable	Input	Integer(10)
3	Error code	I/O	Char(*)

5.6.2.5 Convert to DB (QtmhCvtDB) API

The QtmhCvtDB API provides an interface for CGI programs to parse CGI input, defined as a series of keywords and their values, into a buffer that is formatted according to a DDS file specification. CGI input data, which comes to the CGI program as character data, is converted by the QtmhCvtDB API to the data type defined for the keyword by the corresponding field name in the input DDS file. The required parameters are shown in Table 63.

Table 63. Parameters for the API QtmhCvtDB

Num	Description	Use	Data type
1	Qualified database file name	Input	Char(20)
2	Input string	Input	Char(*)
3	Length of input string	Input	Integer(10)
4	Response variable	Output	Char(*)
5	Length of response variable	Input	Integer(10)
6	Length of response available	Output	Integer(10)
7	Response code	Output	Integer(10)
8	Error code	I/O	Char(*)

5.6.2.6 Parse QUERY_STRING environment variable or Post stdin data (QzhbCgiParse) API

You can use the QzhbCgiParse API to parse the QUERY_STRING environment variable (for the GET method) or standard input (for the POST method) for CGI scripts. If the QUERY_STRING environment variable is not set, the QzhbCgiParse API reads the CONTENT_LENGTH characters from its input. All return output is written to its standard output.

Note

The Command string parameter is used to define a series of flags and modifiers that control how the input data (found in either QUERY_STRING or standard input) should be parsed. These options are defined in *AS/400e HTTP Server for AS/400 Web Programming Guide,* GC41-5435.

The required parameters are shown in Table 64.

Table 64. Parameters for the API QzhbCgiParse

Num	Description	Use	Data type
1	Command string	Input	Char(*)
2	Output format	Input	Char(8)
3	Target buffer	Output	Char(*)
4	Length of target buffer	Input	Integer(10)
5	Length of response	Output	Integer(10)
6	Error code	I/O	Char(*)

5.6.2.7 Produce full HTTP response (QzhbCgiUtils) API

Use the QzhbCgiUtils API to produce a full HTTP 1.0/1.1 response for non-parsed header CGI programs. This API provides functionality similar to the `cgiutils` command used by other IBM HTTP Server platforms. The required parameters are shown in Table 65.

Table 65. Parameters for the API QzhbCgiUtils

Num	Description	Use	Data type
1	Command string	Input	Char(*)
2	Error code	I/O	Char(*)

5.6.3 RPG IV CGI programming

Now we can examine how these APIs are used in an RPG IV program. We have one program, which supports both the GET and POST methods. The only difference between these two methods is in receiving data. The rest of the program is the same for both methods.

5.6.3.1 Introduction to the CGIPROG program

The input data sent from the server to the CGI program is prepared in the name=value format, like this:

```
NAME=Zdravko+Vincetic&CITY=1000+Ljubljana&STREET=Trg+republike+3&
TYPE=P2&SERV=T
```

Special characters used in this string have the following meaning:

=	Links the name and value of input parameter
&	Separates name=value pairs
+	Represents a space (also known as a blank character)

We must use four APIs to perform the following functions:

- QtmhGetEnv to accept input data from the QUERY_STRING variable into the program variable when the GET method is used.
- QtmhRdStin to read input data from the standard input into the program variable when the POST method is used.
- QtmhCvtDb to parse data from the program variable into a data structure with corresponding names and values.
- QtmhWrtStout to write HTML output data to standard output.

Due to the fact that we are using APIs from a service program, the following steps are required to compile the source code and create a program:

1. Enter the PDM option `15` (Create module), or use the command:

   ```
   CRTRPGMOD MODULE(CGIPROG)
   ```

2. Enter the PDM option `26` (Create program), or use the command:

   ```
   CRTPGM PGM(CGIPROG) BNDSRVPGM(QHTTPSVR/QZHBCGI)
   ```

5.6.3.2 Source code for prototypes CGIPROTO

The source member CGIPROTO contains prototypes for calling CGI APIs QtmhGetEnv, QtmhRdStin, QtmhCvtDb, and QtmhWrStout as well as wrapper procedures for these APIs:

```
 * Filename CGIPROTO from CGISRC in RPGISCOOL
 * Prototype for API QtmhGetEnv
 *
D APIEnVar        PR                  ExtProc('QtmhGetEnv')           1
```

```
D  RcvBuffer                 2048A
D  RcvBufferLen                10I 0
D  RspActualLen                10I 0
D  EnvVarName                  64A
D  EnvVarLen                   10I 0
D  ErrorBuffer                 16
 * Prototype for API QtmhRdStin
 *
D APIStdIn        PR                  ExtProc('QtmhRdStin')             2
D  RcvBuffer                 2048A
D  RcvBufferLen                10I 0
D  RspActualLen                10I 0
D  ErrorBuffer                 16
 * Prototype for API QtmhCvtDb
 *
D APICvtDb        PR                  ExtProc('QtmhCvtDb')              3
D  DBFileName                  20A
D  DBBuffer                  2048A
D  DBBufferLen                 10I 0
D  DBStructure                 60
D  DBStructLen                 10I 0
D  DBActualLen                 10I 0
D  DBRespCode                  10I 0
D  ErrorBuffer                 16
 * Prototype for API QtmhWrStout
 *
D APIStdOut       PR                  ExtProc('QtmhWrStout')            4
D  OutBuffer                 2048A
D  OutBufferLen                10I 0
D  ErrorBuffer                 16
 * Prototype for API QCMDEXC
 *
D ExecCommand     PR                  ExtPgm('QCMDEXC')                 5
D  CommandTxt                 256A
D  CommandLen                  15P 5
 * Prototype for procedure #GetEnv
 *
D #GetEnv         PR                                                    6
D  EnvVarName                  64A
D  RcvBuffer                 2048A
D  RspActualLen                10I 0
 * Prototype for procedure #RdStin
 *
D #RdStin         PR                                                    7
D  RcvBuffer                 2048A
D  RspActualLen                10I 0
 * Prototype for procedure #WrStout
 *
D #WrStout        PR                                                    8
D  OutBuffer                 2048A   Value
```

Prototype CGIPROTO notes

1 Prototype for calling CGI API QtmhGetEnv.

2 Prototype for calling CGI API QtmhRdStin.

3 Prototype for calling CGI API QtmhCvtDb.

4 Prototype for calling CGI API QtmhWrStout.

5 Prototype for calling system API QCMDEXC.

6 Procedure #GetEnv requires three input parameters. The sequence of the parameters is changed according to the user preference.

7 Procedure #RdStin requires two input parameters.

8 Procedure #WrStout requires only one input parameter.

5.6.3.3 Source code for the CGIPROG program

Here is the complete source code:

```
 ***********************************************************************
 *  Filename CGIPROG from CGISRC in RPGISCOOL
 *  Simple RPG IV program CGIPROG for both methods GET and POST
 *
 * 1. Compile this source member as module CGIPROG (PDM Option=15)
 *
 * 2. Create program CGIPROG from module CGIPROG (PDM Option=26)
 *    with PROMPT(PF4) and BNDSRVPGM(QHTTPSVR/QZHBCGI)
 ***********************************************************************
 * Order Database file
FOrders    O  A E             Disk    Prefix(X_)
 * HTML  Output file  (prepared HTML Output in SRC-PF HTMLOUT)
FHTMLOut   IF   E             Disk    UsrOpn
 ***********************************************************************
 *
 *-------------------------------------------------------------------
 * 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.
 *-------------------------------------------------------------------
 * Include prototypes
D/Copy RPGISCOOL/CGISRC,CGIPROTO
 *
 * Variables for the CGI interface API QtmhGetEnv and QtmhRdStin
 *
D InBuffer        S           2048A
D InBufLen        S             10I 0 Inz(%Size(InBuffer))
D InActLen        S             10I 0
D EnVarName       S             64A
D EnVarLen        S             10I 0
 * Variables for the CGI interface API QtmhCvtDb
 *
D DBBuff          S           2048A
D DBBuffLn        S             10I 0 Inz(%Size(DBBuff))
D DBDSLn          S             10I 0
D DBActLn         S             10I 0
D DBRespCd        S             10I 0
 * Datastructure INPUT fields
D DBFileName      S             20A   Inz('ORDERS    *LIBL     ')
 * Variables for the CGI interface API QtmhWrStout
 *
D OutBuff         S           2048A
D OutBuffLn       S             10I 0 Inz(%Size(OutBuff))
 * Stand Alone Fields used as work fields
 *
D Command         S            256
D Length          S             15P 5
D PostLenCh       S              5A
D PostLen         S              5S 0
 * Define NewLine and Break
 *
D NewLine         C                   x'15'
D Break           C                   '<BR>'
 * Externally described data structure.  Used for Parsing
 * Need a different one in each CGI-BIN you write
 *
D OrdersDS      E DS                  ExtName(Orders)
 * Data structure for error reporting, from QSYSINC/QRPGLESRC(QUSEC)
 *
D QUSEC           DS
 *                                             Qus EC
D  QUSBPRV                      10I 0 Inz(16)
 *                                             Bytes Provided
D  QUSBAVL                      10I 0
 *                                             Bytes Available
D  QUSEI                         7
 *                                             Exception Id
D  QUSERVED                      1
 *                                             Reserved
 *QUSED01               17    116
 *                                      Varying length
 *-------------------------------------------------------------------
 * Get the Environment Variable called REQUEST_METHOD
 * Set the EnVarName to REQUEST_METHOD
 * Set the EnVarLen to the length of this string
 *
C                   Eval      EnVarName = 'REQUEST_METHOD'
C                   Eval      EnVarLen = %Len(%Trim(EnVarName)) 1
C                   CallP     APIEnVar(InBuffer:InBufLen:InActLen:
C                             EnVarName:EnVarLen:QUSEC)
 * If used HTML method is GET
 *
C                   If        %Subst(InBuffer:1:3) = 'GET'
 *
 * Get the Environment Variable called QUERY_STRING
 * Set the EnVarName to QUERY_STRING
 * Set the EnVarLen to the length of this string
 *
C                   Eval      EnVarName = 'QUERY_STRING' 2
C                   Eval      EnVarLen = %Len(%Trim(EnVarName))
C                   CallP     APIEnVar(InBuffer:InBufLen:InActLen:
C                             EnVarName:EnVarLen:QUSEC)
C                   Else
 *
 * If used HTML method is POST
 *
 * Get the Environment Variable called CONTENT_LENGTH
 * Set the EnVarName to CONTENT_LENGTH
 * Set the EnVarLen to the length of this string
 *
C                   Eval      EnVarName = 'CONTENT_LENGTH' 3
C                   Eval      EnVarLen = %Len(%Trim(EnVarName))
C                   CallP     APIEnVar(InBuffer:InBufLen:InActLen:
C                             EnVarName:EnVarLen:QUSEC)
 * Convert received length from character into number
 *
C                   Eval      PostLenCh = *Zeros
C                   Eval      %Subst(PostLenCh:6-InActLen:InActLen) 4
C                             = %Subst(InBuffer:1:InActLen)
C                   Move      PostLenCh     PostLen
 * Get the input parameters from STDIN using POST method
 *
C                   Z-Add     PostLen       InBufLen
C                   CallP     APIStdIn(InBuffer:InBufLen:InActLen 5
C                             :QUSEC)
C                   EndIf
 *
 * The rest of the program is the same for GET and POST
 * Input data is in InBuffer with length of returned data in InActLen
 * Move this data to the QtmhCvtDb parms for parsing
 *
C                   Eval      DBDSLn = %Size(ORDERSDS)
C                   Eval      DBBuff = %Subst(InBuffer:1:InActLen)
C                   Eval      DBBuffLn = InActLen
 * Parse using the QtmhCvtDb API
 *
C                   CallP     APICvtDB(DBFileName:DBBuff:DBBuffLn:
C                             ORDERSDS:DBDSLn:DBActLn:DBRespCd:QUSEC) 6
 * Field names in external described data structure
 * now contain values passed in input data.
 * Move HTML Form Input data to Database fields
 *
C                   Eval      X_Name = Name 7
C                   Eval      X_City = City
C                   Eval      X_Street = Street
C                   Eval      X_Type = Type
C                   Eval      X_Serv = Serv
 * Write Database record file Orders
 *
C                   Write     OrderR
 *
 * If you had multiple values for the same field, you would have lost
 * all but the one. Cannot reliably predict which one you will get.
 * You need another technique for this situation
 *
 * Override HTMLOut with member OrderHTML and open file
 *
C                   Eval      Command = 'OVRDBF FILE(HTMLOUT) ' + 8
C                             'MBR(ORDERHTML) LVLCHK(*NO) ' +
C                              'OVRSCOPE(*JOB)'
C                   Eval      Length = %Len(%Trim(Command))
C                   CallP     ExecCommand(Command:Length)
C                   Open      HTMLOut
 *
 * Create the HTML Output in OutBuff field
 * Write HTML Required control records
 * ADD NewLine append after 80 to 120 characters to OutBuff
 *
C                   Do        9            9
C                   Read      HTMLOut
C                   Eval      OutBuff = %Trimr(OutBuff) + %Trimr(SrcDta)
C                             + NewLine
C                   EndDo
C                   Eval      OutBuff = %Trim(OutBuff) + Break + Break
C                             + %Trimr(Name) + Break
C                             + %Trimr(City) + Break + %Trimr(Street)
C                             + Newline + Break + Break
C                   Select
C                   When      Type = 'P1'
C     10            Chain     HTMLOut
C                   When      Type = 'P2'
C     11            Chain     HTMLOut
C                   When      Type = 'P3'
C     12            Chain     HTMLOut
C                   EndSl
C                   Eval      OutBuff = %Trimr(OutBuff) + %Trimr(SrcDta)
C                             + NewLine
C                   If        Serv = 'T'
C     13            Chain     HTMLOut
C                   Else
C     14            Chain     HTMLOut
C                   EndIf
C                   Eval      OutBuff = %Trimr(OutBuff) + %Trimr(SrcDta)
C                             + NewLine
C     15            Setll     HTMLOut
C                   Read      HTMLOut
C                   Dow       Not %Eof(HTMLOut)
C                   Eval      OutBuff = %Trimr(OutBuff) + %Trimr(SrcDta)
C                             + NewLine
C                   Read      HTMLOut
C                   EndDo
 *
 * Send OutBuff to standard output
 *
C                   Eval      OutBuffLn = %Len(%Trimr(OutBuff))
C                   CallP     APIStdOut(OutBuff:OutBuffLn:QUSEC) 10
 *
 * End program
 *
C                   Close     HTMLOut 11
C                   Eval      Command = 'DLTOVR FILE(HTMLOUT) LVL(*JOB)'
C                   Eval      Length = %Len(%Trim(Command))
C                   CallP     ExecCommand(Command:Length)
C                   Eval      *InLR = *On
```

5.6.3.4 Explanation of the CGIPROG program

The program uses two files: the order file ORDERS where accepted orders are written, and the HTMLOUT file, which contains HTML data used to prepare HTML output document.

The file ORDERS has the following definition:

```
A***************************************************************
A* Physical file ORDERS
A***************************************************************
A          R ORDERR
A            NAME          40A         COLHDG('Customer Name')
A            CITY          40A         COLHDG('City')
A            STREET        40A         COLHDG('Street')
A            TYPE           2A         COLHDG('Type')
A            SERV           1A         COLHDG('Service')
```

In this program, we use the PREFIX keyword with this file to temporarily change field names by adding the prefix "X_". This is required because the same names are also used to convert data from the input string received from the Web browser.

The file HTMLOUT is a source physical file, where the member ORDERHTML contains data required to create an output HTML document, which is sent to the Web user as a confirmation for their order. The numeration is not part of the data, and is added only for easier understanding of the program logic.

```
01 - content-type: text/html
02 -
03 - <HTML><HEAD>
04 - <TITLE>Ordering an AS/400</TITLE>
05 - </HEAD>
06 - <BODY bgcolor="#F8F8FF">
07 - <IMG align=middle src="/BonusImg/adserver.gif">
08 - <H1>Ordering an IBM System AS/400</H1>
09 - Thanks for ordering an IBM System AS/400 :
10 - Model: Portable            <BR>
11 - Model: Server 50 S         <BR>
12 - Model: System              <BR>
13 - Support Line Service : YES <BR>
14 - Support Line Service : NO  <BR>
15 - <P>
16 - <A HREF="/democgi">Back</A> Order demo
17 - </BODY></HTML>
```

To access this member in the source file, we use the Override Database File (OVRDBF) command, which is executed by calling the system API QCMDEXC. In the same way, we run the Delete Override (DLTOVR) command at the end of the program.

On D specifications, we first copy the source member CGIPROTO to include the prototype definitions for calling CGI APIs and system API QCMDEXC. Then, we define required data fields used by these APIs.

ORDERSDS is an externally described data structure, which used for parsing received input data. It has the same record format as the database file ORDERS. Therefore, we use the keyword EXTNAME. This is also the reason why we need the PREFIX keyword with the file. It is used by QtmhCvtDb API to parse and convert data from the name=value format received from QtmhGetEnv or QtmhRdStin API to separate fields with corresponding names.

The program logic includes the following actions:

1 Using API QtmhGetEnv, we retrieve the environment variable REQUEST_METHOD, which contains method from the Form document, either GET or POST.

2 If the requested method is GET, we use again the API QtmhGetEnv to retrieve the environment variable QUERY_STRING, which contains the input data.

3 If the requested method is POST, we first use the API QtmhGetEnv to determine the length of the input string by retrieving the environment variable CONTENT_LENGTH.

4 The content of the environment variable CONTENT_LENGTH is a character string, which must be extracted and converted to numeric field.

5 Finally, we run the API QtmhRdStin to retrieve input data in the given length from the standard input.

6 The rest of the program is the same for both methods. First, we use the API QtmhCvtDb to parse the name=value pairs from the input string into the corresponding data structure, with the same field names as the input string.

7 Now the data can be moved into database fields, and the record is written to the file ORDERS.

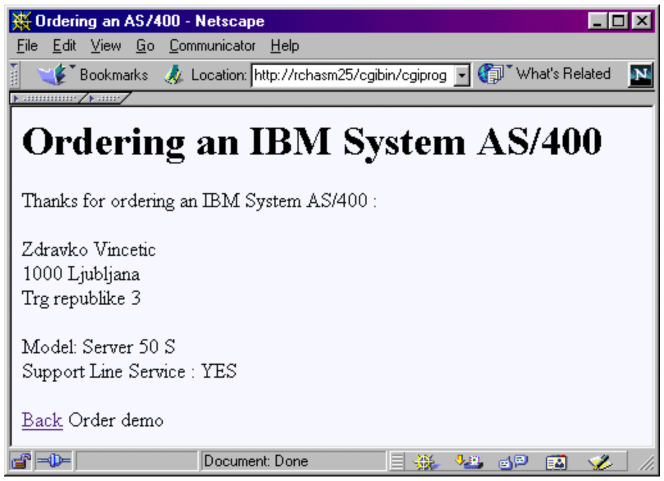
8 To access data from the source file member, we must run the OVRDBF command using the system API QCMDEXC. Then, we can open the file.

9 The HTML output is prepared. It reads the file HTMLOUT and combs records from this file with newline and break constants.

10 To send output data to standard output, we use the API QtmhWrStout.

11 At the end of the program, we close the file HTMLOUT and call the system API QCMDEXC to delete the database override definition.

The HTTP server forwards the prepared HTML document to the browser. The result of our program is shown in Figure 32.



Ordering an AS/400 - Netscape
File Edit View Go Communicator Help
Bookmarks
Location: http://rchasm25/cgibin/cgiprog
What's Related
Document: Done

Ordering an IBM System AS/400

Thanks for ordering an IBM System AS/400 :

Zdravko Vincetic
1000 Ljubljana
Trg republike 3

Model: Server 50 S
Support Line Service : YES

Back Order demo

Figure 32. The HTML output document displayed by the Web browser

Try it yourself

You can try this example by compiling the code from this section on your AS/400 system. Use the following commands to create the program:

```
ADDLIBLE LIB(RPGISCOOL)
CRTRPGMOD MODULE(RPGISCOOL/CGIPROG) SRCFILE(RPGISCOOL/CGISRC)
CRTPGM PGM(RPGISCOOL/CGIPROG) BNDSRVPGM(QHTTPSVR/QZHBCGI)
```

With the WRKHTTPCFG command, change the HTTP server configuration, and add the following directives to enable the CGI program:

```
Pass  /democgi   /QSYS.LIB/RPGISCOOL.LIB/CGISRC.FILE/CGIHTML.MBR
Exec  /cgibin/*  /QSYS.LIB/RPGISCOOL.LIB/*.PGM
```

To run the program, start the Web browser, and enter the following URL:

```
http://your-server-name/democgi
```

5.6.4 Simplifying CGI programming

API calls from a CGI program can be simplified by creating your own procedures to reduce the complexity and number of required parameters. These procedures should be placed in a separate service program, which is then bound to our CGI program.

We provide you an example of such a service program, which contains the following procedures:

- #GetEnv to call API QtmhGetEnv
- #RdStin to call API QtmhRdStin
- #WrStout to call API QtmhWrStout

The API QtmhCvtDb requires detailed information about the field names used within the HTML Form tag. These input field names (as shown in 5.6.1, “HTML form document” on page 236) are specific for each program and cannot be included in a general purpose service program.

To compile and create this service program, the following steps are required:

1. Enter the PDM option `15` (Create module), or enter the command:

   ```
   CRTRPGMOD MODULE(CGISERV)
   ```

2. Enter the PDM option `27` (Create service program), or enter the command:

   ```
   CRTSRVPGM SRVPGM(CGISERV) EXPORT(*ALL) + BNDSRVPGM(QHTTPSVR/QZHBCGI)
   ```

5.6.4.1 Source code for the service program CGISERV

Here is the complete source code:

```
***********************************************************************
 *  Filename CGISERV from CGISRC in RPGISCOOL
 *  Service program CGISERV to simplify API calls
 *
 * 1. Compile this source member as module CGISERV (PDM Option=15)
 *
 * 2. Create serv. program CGISERV from module CGISERV (PDM Option=27)
 *    with PROMPT(PF4) and BNDSRVPGM(QHTTPSVR/QZHBCGI)
 ***********************************************************************
H NoMain                                                                1
 ***********************************************************************
 *
 * Include prototypes
D/Copy RPGISCOOL/CGISRC,CGIPROTO                                        2
 *
 * Global field definitions
 *
D QUSEC           DS                                                    3
D  QUSBPRV                      10I 0 Inz(%size(QUSEC))
D  QUSBAVL                      10I 0
D  QUSEI                         7
D  QUSERVED                      1
 *QUSED01                17    116
 *---------------------------------------------------------------
 * Procedure #GetEnv - Get Environment Variable
 *
P #GetEnv         B                   Export                            4
D #GetEnv         PI
D  VarName                      64A
D  InData                     2048
D  RespLen                      10I 0
 *
D  InDataLen      S             10I 0
D  VarNameLen     S             10I 0
 *
C                   Eval      InData = *Blanks
C                   Eval      InDataLen = %Len(InData)
C                   Eval      VarNameLen = %Len(%Trim(VarName))
```

```
C                   CallB     'QtmhGetEnv'
C                   Parm                    InData
C                   Parm                    InDataLen
C                   Parm                    RespLen
C                   Parm                    VarName
C                   Parm                    VarNameLen
C                   Parm                    QUSEC
P #GetEnv         E
 *---------------------------------------------------------------
 * Procedure #RdStin - Read Standard Input
 *
P #RdStin         B                   Export                           5
D #RdStin         PI
D  InData                     2048
D  RespLen                      10I 0
 *
D  InDataLen      S             10I 0
 *
C                   Eval      InData = *Blanks
C                   Eval      InDataLen = RespLen
C                   CallB     'QtmhRdStin'
C                   Parm                    InData
C                   Parm                    InDataLen
C                   Parm                    RespLen
C                   Parm                    QUSEC
P #RdStin         E
 *---------------------------------------------------------------
 * Procedure #WrStout - Write Standard Output
 *
P #WrStout        B                   Export                           6
D #WrStout        PI
D  OutData                    2048    Value
 *
D  OutDataLen     S             10I 0
 *
C                   Eval      OutDataLen = %Len(%Trim(OutData))
 *
C                   CallB     'QtmhWrStout'
C                   Parm                    OutData
C                   Parm                    OutDataLen
C                   Parm                    QUSEC
P #WrStout        E
```

Program CGISERV notes

1 Keyword NOMAIN at H specifications improves the program performance, and can be used if the program does not contain the main procedure.

2 Copy prototype definitions from the member CGIPROTO.

3 Error data structure is used in all procedures, and therefore is defined as a global field definition.

4 Procedure #GetEnv requires three input parameters. The sequence of the parameters is changed according to user preference.

5 Procedure #RdStin requires two input parameters.

6 Procedure #WrStout requires only one input parameter.

5.6.4.2 Using service program CGISERV

The following code snippet illustrates how to call our subprocedures from a CGI program to invoke the required API functions. Do not forget that subprocedures must be prototyped each time you use them. We recommend that you put prototypes for all these procedures into a separate source member, as we did, which can be copied into all CGI programs and the service program itself. This ensures that the same prototypes are always used.

```
**********************************************************************
 * Order Database file
FOrders    O  A E             Disk    Prefix(X_)
 * HTML  Output file  (prepared HTML Output in SRC-PF HTMLOUT)
FHTMLOut   IF   E             Disk    UsrOpn
```

```
 *******************************************************************
 *
 * Include prototypes
D/Copy RPGISCOOL/CGISRC,CGIPROTO
 *
 *------------------------------------------------------------------
 * Call API QtmhGetEnv
 *
C                   Eval      EnVarName = 'QUERY_STRING'
C                   CallP     #GetEnv(EnVarName : InBuffer : InActLen)
 * Call API QtmhRdStin
 *
C                   Z-Add     PostLen       InActLen
C                   CallP     #RdStin(InBuffer : InActLen)
 * Call API QtmhWrStout
 *
C                   CallP     #WrStout(OutBuff)
```

5.6.5 Persistent CGI

A persistent CGI is an extension to the CGI interface that allows a CGI program to remain active across multiple browser requests and maintain a session with that browser client. The session is actually disconnected when the response is sent to the Web client, and then connected again when a new request comes from the same client. This allows files to be left open, the state to be maintained, and complex database transactions to be committed or rolled-back based on user input.

The AS/400 CGI program must be written using named activation groups. This allows the program to remain active after returning. The CGI program notifies the server that it wants to remain persistent by using the "Accept-HTSession" CGI header as the first header it returns. This header defines the session ID associated with this instance of the CGI program and is not returned to the browser. Subsequent URL requests to this program must contain the session ID, which should be written as the first value after the program name. This is called pathinfo, and this value is presented to the program as an environment variable PATH_INFO.

The server uses this ID to route the request to that specific instance of the CGI program. It is possible to reuse an existing session ID, which enables the use of a back button in the Web browser. The CGI program could, if necessary, regenerate this session ID for each request. We strongly recommend that you use Secure Sockets Layer (SSL) for persistent and secure business transaction processing.

For additional information, see *HTTP Server for AS/400 Web Programming Guide*, GC41-5435.

5.6.6 More information on CGI programming in RPG IV

Before you start writing your own CGI programs, we highly recommend that you look at the AS/400 Internet site, where you can find a library of code examples, samples, and tools. Go to the following address on the Web:

`http://www.as400.ibm.com/snippets`

First, select **View readme file**, followed by **View CGIDEVDH**. The document describes the library CGIDEV, which contains a CGI tools service program and a sample template program, which illustrates how to use the tools. Most of the high-level language programming in CGI is written in RPG IV. There is little RPG and CL.

This library can be downloaded and installed on your AS/400 system. As a result of downloading the library, you get a savefile, from which you can restore the CGIDEV library. You can use this information for writing you CGI applications.

There is an advanced version of all these tools, available for a fee from the IBM AS/400 Custom Technology Center, which part of the IBM Rochester AS/400 Laboratory. Their experienced AS/400 developers provide software development services for AS/400 solutions. You can find them on the Internet at:
`http://www.as400.ibm.com/service/welcome_3.htm`

For more information, refer to the following pages on the Web:

- `http://www.as400.ibm.com/snippets`
  1. Select **View Readme files**, followed by **View CGIDEVDH**.
  2. Select RPG in the drop-down list.
- `http://www.as400.ibm.com/developer/ebiz/cgi/`
- `http://www.easy400.ibm.it`
- `http://hoohoo.ncsa.uiuc.edu/cgi/examples.html`

Also, refer to the manual *AS/400e HTTP Server for AS/400 Web Programming Guide,* GC41-5435.

5.7 Understanding UNIX-POSIX APIs through IFS examples

This section discusses using UNIX or POSIX type APIs by using the Integrated File System (IFS) application program interfaces (APIs) as an example. If you tried to use any of the UNIX or POSIX APIs in the past, you may have quickly noticed that all the examples in the reference manuals use the C language. Since this is an RPG IV book, we demonstrate that it is far easier for you to use RPG IV.

This section contains RPG IV programming examples using the two generations of IFS APIs. The distinction between the two generations is the use of a path name structure, which differentiates the new APIs than the old one, where only the path name needs to be specified. More details on the path name structure are also explained, as well as an IFS introduction.

The first generation of IFS APIs shown in this section are the APIs used to perform basic operations on stream files, such as create, open, read, write, seek, and close stream files (see 5.7.4, “Introducing basic stream file APIs” on page 256). Through a complete set of examples, we show you how to create a new stream file, write in it, and read from it, plus use a seek function to change the cursor location within the file. The RPG IV code required to use them is explained with the coding examples and a complete sample program, which merges stream file text with variables values. The complete code for the sample program using RPGIV subprocedure and pointers programming techniques among others is also included and explained through an outline summary.

The second generation of the IFS APIs is also explained in this section through a sample program using the Qp0lProcessSubtree API (see 5.7.5, “Using more complex IFS APIs: Qp0lProcessSubtree()” on page 280). This API scan any IFS directories and call a user created exit program for any objects found matching the selection criteria specified by the RPG IV calling program. This example uses an external subprocedure as the exit program and shows how to define an array of pointers in RPG IV, among other programming techniques. The complete sample code of a scanning directory application is also included and explained through an outline summary.

The programming examples use the C runtime function to materialize any error reported by the IFS APIs. This technique is also explained at the end of this section (see 5.7.6, “IFS APIs error reporting” on page 301). Also, we use a user created command with list type parameters in these examples to pass the input parameters to our main procedure. You can also find coding examples retrieving data out of those more complex parameters.

Displaying and editing stream files

As you work through the examples in this section, you may want the ability to edit and display the stream files located in the IFS. IBM has provided PTFs that go back to V3R7M0 that contain two new commands: Edit File (EDTF) and Display Stream File (DSPSTMF). Refer to the following list when ordering the appropriate PTF for your system:

- SF49052 for V4R3M0
- SF45296 for V4R2M0
- SF41518 for V4R1M0
- SF38832 for V3R7M0

In V4R4M0, the Edit File (EDTF) command made it into OS/400. If you still need the Display Stream File (DSPSTMF) command, it can be found in SF55871.

5.7.1 The Integrated File System (IFS)

The IFS is a part of OS/400 that lets you support stream input/output and storage management similar to personal computer and UNIX operating systems, while providing a structure for all information stored in the AS/400 system.

You will find an introduction to the IFS in Appendix B, “An introduction to the Integrated File System (IFS)” on page 413.

5.7.2 The API manual

The APIs that perform operations on IFS directories and stream files are in the form of C language functions. With RPG IV, you can use the IFS C APIs that are included in OS/400 to perform operations on directories, files, and related objects in the file systems accessed through the IFS interface.

As you may know already, there is no API reference manual entitled *"*IFS APIs*"*. The APIs related to the IFS are documented in the *System API Reference OS/400 UNIX-Type APIs,* SC41-5875, in the chapter "Integrated File System APIs". They are described under the UNIX-Type APIs manual as. Being C functions, they are based on UNIX standards.

ILE C/400 functions

ILE C/400 provides the standard C functions defined by the American National Standards Institute (ANSI), which also support the IFS stream I/O. These APIs, such as fopen(), are also available to be used by an RPG IV application to access DB2/400 files as stream files, but cannot be used to access IFS files directly. Only the ILE C/400 compile can reroute functions such as fopen() to the IFS.

If you want to use RPG IV, you can still use these C functions to access IFS files by manually adding the "_C_IFS_" prefix in front of the function. For example, "fopen" will become "_C_IFS_fopen". You need to specify the QC2IFS binding directory when creating your program.

5.7.3 The path name

The path name format is common across application programming interfaces that work with objects, which are supported across file systems. These APIs require a path name to identify the object with which the API will work.

When using an IFS API to operate on an object, you identify the object by supplying its directory path, following the path name rules specific to the file system where the IFS object is located. For more information on path name rules and the IFS, refer to Appendix B, “An introduction to the Integrated File System (IFS)” on page 413.

As previously noted, the older APIs do not use the path name structure. They accept the path name as a character string and rely on the information associated with the job to determine which CCSID, country ID, or language ID to use.

The trend with the newer APIs is to use the path name format or structure, which contains the CCSID, Country ID and language ID, among others, and expects the path name to be coded using the value specified in those parameters. For example, most of the new APIs, but not all of them, pass path names to their exit programs using the Unicode (UCS-2) CCSID, even if the job uses a different one. In this case, the CCSID parameter is useful to indicate under which CCSID this path name has been coded.

The APIs not using the Path name structure usually require the path name string passed to the API as a parameter to be null terminated (x’00’). This extra step is required because of the C language origins of these APIs, but can be automated by using the OPTION(*STRING) keyword on the prototype parameter definition.

5.7.3.1 Path name structure or path name format

This structure defined the path name characteristics, such as:

- Code character set ID (CCSID)
- Country ID
- Language ID
- Path Type Indicator
- Length of path name
- Path name delimiter character

Plus, it contains the path name itself, as a variable field length or as a pointer, at the last position in the structure. When the path name is passed as a variable field length, the path name does not need to be null terminated (x'00').

The API documentation refers to the Qlg_Path_Name_T format to indicate that the path name must be passed to (or will be passed by) the API using the path name structure. The path name structure is provided as a reference in the QSYSINC library, which is part of the OS/400 - System Openness Includes (57xxSS1 - option 13) licensed program product. Here is an abstract of the member QLG, in file QRPGLESRC, which contains the following sample data structure definition for the path name structure:

```
D*****************************************************************
D*Structure for NLS enabled path name
D****
D*NOTE: The following type definition only defines the fixed
D*  portion of the format. Any varying length field will
D*  have to be defined by the user.
D*****************************************************************
DQLGPN            DS
D*                                           Qlg Path Name
D QLGCCSID02              1      4B 0
D*                                           CCSID
D QLGCID                  5      6
D*                                           Country ID
D QLGLID                  7      9
D*                                           Language ID
D QLGERVED07             10     12
D*                                           Reserved
D QLGPT                  13     16B 0
D*                                           Path Type
D QLGPL                  17     20B 0
D*                                           Path Length
D QLGPND                 21     22
D*                                           Path Name Delimiter
D QLGRSV200              23     32
D*                                           Reserved2
D*QLGPN00                33     33
D*
D*                                  Variable length field
```

This data definition is the result of an automatic conversion from the original C language include files. Because this member (QSYSINC/QRPGLESRC.QLG) contains others data definition, we recommend that you do not /COPY the full contents of it in your programs. As you can see from the above definition, the last field (QLGPN00) should contain the path name. However, it has been commented out since it is definition can either be a pointer or a character field using various lengths.

We recommend that you create your own version of the QLGPN or Qlg_Path_Name_T format, which would contain only permanent entry of the data structure. You can use the /COPY member function to include them into your program, which would declare the Data Structure heading, plus the path name field definition. As example, the Path name structure definition in your program would look like this:

```
 * Path name structure definition
DQLGPN            DS
 /COPY RPGISCOOL/SWEEPSRC,PATHNAMEDF
D QLGPN00                      8000A
```

The /COPY member PATHNAMEDF would be:

```
 * Path name structure definition based on QSYSINC/QRPGLESRC.QLG
 *  member name: PATHNAMEDF
D QLGCCSID02                     10U 0
 *                                      CCSID
```

```
D QLGCID                          2
 *                                         Country_ID
D QLGLID                          3
 *                                         Language_ID
D QLGERVED07                      3
 *                                         Reserved
D QLGPT                          10U 0
 *                                         Path_Type
 *                                         0=QLGPN00 is a path name
 *                                         1=QLGPN00 is a pointer
 *                                         2=same as 0 with UNICODE
 *                                         3=same as 1 with UNICODE
D QLGPL                          10U 0
 *                                         Path_Length
D QLGPND                          2
 *                                         Path_Name_Delimiter
D QLGRSV200                      10
 *                                         Reserved2
```

As you see, we did not include the Data Structure (DS) definition instruction in the /COPY member. The keyword BASED(pointer) can be used in some circumstances to map the structure to a specific memory location using a pointer, for example:

```
 * Path name structure definition
DQLGPN           DS                  Based(PathName@)
 /COPY RPGISCOOL/SWEEPSRC,PATHNAMEDF
D QLGPN00                          *
```

Some of the path name characteristics, such as the path name delimiter, character set, country ID, or language ID may be constant on your system. You can also choose to initialize those values in the /COPY member instead of in every procedure that is using it.

For more information on the path name structure, refer to the AS/400 Information Center at: `http://www.as400.ibm.com/infocenter`

Once inside the Information Center site, select **Programming**, and then **OS/400 APIs**. You should find **Path name structure** under **Concepts**.

5.7.4 Introducing basic stream file APIs

This section demonstrates the usage of some IFS APIs on such stream file basic operations as stream file creation, open, read, write and close. Using coding examples, we created a small sample program, which creates a stream file, and a second one, which writes data into it. The main sample program, MERGESTMF, merges text with a master file containing variable to produce a new stream file.

The following section contains:

- A sample program, CREATSTMF, creating a stream file, introducing the open() and close() APIs.
- A second sample program, WRITESTMF, writing to a stream file, using the write() API along with the two APIs used in the first program.
- Description of the read() and lseek() APIs used to read a stream file content and change the file offset location.
- An Overall idea of the *MERGESTMF* main sample program, the anatomy of the program, including an outline and the complete sample code.

5.7.4.1 Creating, opening, and closing a stream file object

This section explains the usage of the IFS APIs used to open, create, and close a stream file. As explained later, we are using the open() API to create the stream file.

The list of IFS APIs used are:

- open() — Open file
- close() — Close file

The sample program CREATSTMF helps you understand the usage of those APIs to create or clear a stream file, and then close it. All those APIs are documented in the *System API Reference OS/400 UNIX-Type APIs,* SC41-5875, in the chapter "Integrated File System APIs".

Note: None of the following APIs required the use of the path name structure described earlier.

In the following sections, you can correlate the marker of each API example to the main source code in 5.7.4.6, "Sample program code" on page 273.

open() — Open file

The open() function opens a file and returns a number called a *file descriptor*. You can use this file descriptor to refer to the file in subsequent I/O operations such as read() or write(). Each file opened by a job gets a new file descriptor. Optionally, this API can also create a new file if the file specified does not exist and the proper file status flag has been specified.

Using the creat() function

The creat() function also creates a new file or rewrites an existing file so that it is truncated to zero length. We recommend that you use the open() function instead of the creat() function since the open() function allows you to specify a specific codepage different than the default one. Plus, you need to reuse the same API anyway to reopen the file for I/O processing.

The following C syntax can be used as a reference for prototyping parameters required to call this function:

```
# include <fcntl.h>
int open(const char *path, int oflag, . . .);
```

Table 66 shows the parameters for this function. These parameters must be defined in the prototype used to call this API.

Table 66. Parameters for the open() function

Argument	Description	Use	RPG data type	C data type
	Return value	Output	Integer(10)	int
path	Path name	Input	Pointer	char *
oflag	file access mode, status flags and share mode	Input	Integer(10)	int
mode	File mode or file permission bits	Input optional	Numeric, unsigned(10)	mode_t

Argument	Description	Use	RPG data type	C data type
codepage	Codepage	Input optional	Numeric, unsigned(10)	unsigned

Here is the prototype used for the open API in an RPG IV format:

```
 *  IFSPROTO from IFSSRC in RPGISCOOL (part 1 of 5)
 * open function prototype                                  2
D open            PR            10I 0 ExtProc('open')
D  path@                          *   Value options(*string)
D  oflag                        10I 0 Value
D  mode                         10U 0 Value Options(*nopass)
D  codepage                     10U 0 Value Options(*nopass)
```

The different parameters are discussed in the following sections. The sample program CREATSTMF shows where we defined the parameters used by the API, the return value field, and the call statement (see marker 2, 4, and 7). Please refer to the following sections for explanations of the other markers.

Open file parameter: Return value

The return value returned is the file descriptor associated with the open file. This file descriptor is used by the other IFS APIs to refer to this file. In case a problem occurred during the file opening, the return value will contain "-1", and the *errno* global variable is set to indicate the error.

The *errno* value can be retrieved using various C APIs such as: perror(), __errno(), and strerror(). These APIs are explained in 5.7.6, “IFS APIs error reporting” on page 301.

Open file parameter: Path name

The path name field of the file to be opened is passed to the API as an expression by using the OPTION(*STRING) keyword on the prototype parameter definition. This path name must refer to a character field name where the maximum length is within the file system limits. The OPTION(*STRING) keyword also takes care of having the path name terminated by a null character (x’00’), as required by the API.

Open file parameter: File access mode, status flag, and share mode

The file access mode, status flags, and share mode descriptions are used to define how the file will be opened (read, write or read/write), and with which attributes (append, create, truncate and so on). The C definition of each flag can be found in the member FCNTL of the source file QSYSINC/H from the system openness include licensed program product. The following file sample includes the C definition of the file access mode, status flags, and share mode as comments with the equivalent definition of those fields in an RPG IV format on the next line. We also converted the value from the Octal notation to hexadecimal:

```
 * OPENDFN from IFSSRC in RPGISCOOL (part 1 of 2)
 * structure definition for open() function                   4
 *     from QSYSINC/H.FCNTL member
 */*******************************************************************/
 */*   File Access Modes                                           */
 */*******************************************************************/
 *#define O_RDONLY   00001        /* Open for reading only          */
D O_RDONLY        S             10I 0 INZ(x'01')
 *#define O_WRONLY   00002        /* Open for writing only          */
D O_WRONLY        S             10I 0 INZ(x'02')
```

```
 *#define O_RDWR     00004        /* Open for reading and writing    */
D O_RDWR          S             10I 0 INZ(x'04')
 *#define O_CREAT      00010      /* Create file if it doesn't exist */
D O_CREATE        S             10I 0 INZ(x'08')
 *#define O_EXCL       00020      /* Exclusive use flag              */
D O_EXCL          S             10I 0 INZ(x'10')
 */*                00040           reserved                        */
 *#define O_TRUNC      00100      /* Truncate flag                   */
D O_TRUNC         S             10I 0 INZ(x'40')
 */*****************************************************************/
 */*   File Status Flags values                                    */
 */*****************************************************************/
 *#define O_CODEPAGE   040000000  /* code page flag                  */
D O_CODEPAGE      S             10I 0 INZ(x'800000')
 *#define O_TEXTDATA  0100000000  /* text data flag                  */
D O_TEXTDATA      S             10I 0 INZ(x'01000000')
 *#define O_APPEND     00400      /* Set append mode                 */
D O_APPEND        S             10I 0 INZ(x'0100')
 *#define O_LARGEFILE  004000000000  /* Large file access            */
D O_LARGEFILE     S             10I 0 INZ(x'20000000')
 *#define O_INHERITMODE 001000000000 /* inherit mode flag            */
D O_INHERITMODE   S             10I 0 INZ(x'08000000')
 */*****************************************************************/
 */*   File Share Mode Values                                      */
 */*****************************************************************/
 *#define O_SHARE_RDONLY 000000200000 /* Share with readers only     */
D O_SHARE_RDONLY  S             10I 0 INZ(x'010000')
 *#define O_SHARE_WRONLY 000000400000 /* Share with writers only     */
D O_SHARE_WRONLY  S             10I 0 INZ(x'020000')
 *#define O_SHARE_RDWR   000001000000 /* Share with readers and
 *                                      writers                     */
D O_SHARE_RDWR    S             10I 0 INZ(x'040000')
 *#define O_SHARE_NONE   000002000000 /* Share with neither readers
 *                                      nor writers                 */
D O_SHARE_NONE    S             10I 0 INZ(x'080000')
```

Even if those fields are used as a constant within the program referring to them, we had to declare them as a standalone field (S) because a hexadecimal expression initializes a constant field as a character field instead of a numeric field. The flag description can be found in the documentation of the open() API in the manual *System API Reference OS/400 UNIX-Type APIs*, SC41-5875.

Though possibly additional static storage is created, we recommend that you /COPY this member file into your RPG program for ease of use and comprehension of those flags. Flags can be added to each other when multiple flags are required such as on the same open instructions (see marker 6 and 7).

Open file parameter: Security attributes or permission bits (optional)
The security attributes or permission bits are used to specify the security attributes of the stream file (authorities). This optional parameter must only be used when the file status flag contains the O_CREAT or O_CODEPAG values. When O_CREAT hasn't been specified on the file status flag (for example where only O_CODEPAG was specified), this keyword has no effect. However, something needs to be specified so the *Codepage* parameters can be read (prototype *NOPASS restriction).

The C definition of each modes can be found in the member STAT of source file QSYSINC/SYS from the System Openness includes licensed program. The following file sample includes the C definition of the security attributes as comments with the equivalent definition of those in an RPG IV format, converted from the octal notation to hexadecimal:

```
 * OPENDFN from IFSSRC in RPGISCOOL (part 2 of 2)
 * from QSYSINC/SYS.STAT member                                4
 */*******************************************************************/
 */* Definitions of Security Attributes and File Types                */
 */*******************************************************************/
 * Owner attributes
 *   #define S_IRUSR 0000400    /* Read for owner                    */
D S_IRUSR         S             10I 0 INZ(x'0100')
 *   #define S_IWUSR 0000200    /* Write for owner                   */
D S_IWUSR         S             10I 0 INZ(x'80')
 *   #define S_IXUSR 0000100    /* Execute and Search for owner      */
D S_IXUSR         S             10I 0 INZ(x'40')
 *   #define S_IRWXU (S_IRUSR|S_IWUSR|S_IXUSR)  /* Read, Write,
 *                                               Execute for owner   */

 * Primary group attributes
D S_IRWXU         S             10I 0 INZ(x'01C0')
 *   #define S_IRGRP 0000040         /* Read for group               */
D S_IRGRP         S             10I 0 INZ(x'20')
 *   #define S_IWGRP 0000020         /* Write for group              */
D S_IWGRP         S             10I 0 INZ(x'10')
 *   #define S_IXGRP 0000010         /* Execute and Search for group  */
D S_IXGRP         S             10I 0 INZ(x'08')
 *   #define S_IRWXG (S_IRGRP|S_IWGRP|S_IXGRP)  /* Read, Write,
 *                                               Execute for group   */
D S_IRWXG         S             10I 0 INZ(x'38')

 * *PUBLIC attributes
 *   #define S_IROTH 0000004         /* Read for other               */
D S_IROTH         S             10I 0 INZ(x'04')
 *   #define S_IWOTH 0000002         /* Write for other              */
D S_IWOTH         S             10I 0 INZ(x'02')
 *   #define S_IXOTH 0000001         /* Execute and Search for other  */
D S_IXOTH         S             10I 0 INZ(x'01')
 *   #define S_IRWXO (S_IROTH|S_IWOTH|S_IXOTH)  /* Read, Write,
 *                                               Execute for other   */
D S_IRWXO         S             10I 0 INZ(x'07')
```

Even if those fields are used as a constant within the program referring to them, we had to declare them as a standalone field (S) because a hexadecimal expression initializes a constant field as a character field instead of a numeric field. The flag description can be found in the documentation of the chmod(), Change File authorizations API, in the manual *System API Reference OS/400 UNIX-Type APIs*, SC41-5875.

Although additional static storage may be created, we recommend that you /COPY this member file into your RPG program for ease of use and comprehension of those flags.

The following tables can be use to compare the security attribute flag and their equivalence with the OS/400 stream file data authorities.

Stream file authorities

The *usage note* section in the open() API documentation contains important information on each file system specifications and their own ways of handling authorities. More information on OS/400 security can be found in *Security - Reference*, SC41-5302.

The different permission bits for the file owner are shown in Table 67.

Table 67. Data authority for the file owner

Argument	Description	Decimal Value	Data authority
S_IRUSR	Read permission for the file owner	256	*R
S_IWUSR	Write permission for the file owner	128	*W

Argument	Description	Decimal Value	Data authority
S_IXUSR	Search permission (for a directory) or execute permission (for a file) for the file owner	64	*X
S_IRWXU	Read, write, and search or execute for the file owner; bitwise inclusive OR of S_IRUSR, S_IWUSR, and S_IXUSR	448	*RWX

Table 68 lists the different authorities available for the primary group of the stream file.

Table 68. Data authority for the primary group

Argument	Description	Decimal value	Data authority
S_IRGRP	Read permission for the file's group	32	*R
S_IWGRP	Write permission for the file's group	16	*W
S_IXGRP	Search permission (for a directory) or execute permission (for a file) for the file's group	8	*X
S_IRWXG	Read, write, and search or execute permission for the file's group; bitwise inclusive OR of S_IRGRP, S_IWGRP, and S_IXGRP	56	*RWX

Table 69 shows the different permissions available for the *public authority keyword.

*Table 69. Data authority for *public*

Argument	Description	Decimal Value	Data authority
S_IROTH	General read permission	4	*R
S_IWOTH	General write permission	2	*W
S_IXOTH	General search permission (for a directory) or general execute permission (for a file)	1	*X
S_IRWXO	General read, write, and search or execute permission; bitwise inclusive OR of S_IROTH, S_IWOTH, and S_IXOTH	7	*RWX

Flags from the same or different elements (owner, group or *public) can be added to each other when multiple flags are required on the same open instruction, as described in the sample code (see marker 7).

In the previous sample program, a file created using these mode parameters has the following authorities:

Owner	Read, write, and execute authorities on the object (*RWX).
Group	Read, write, and execute authorities on the object (*RWX).
***PUBLIC**	Read and execute authority (*RX)

Codepage (optional)

This optional parameter indicates under which codepage the date in the stream file will be kept if used during the file creation. Or, it may indicate that the codepage of the data read or write to the file, on a regular file open on an already existing file.

The flag O_CODEPAGE must have been specified on the file status flag parameter for this parameter to be used by the API. Also a value must have been specified on the Security attribute parameter so this keyword can be read by the API. This is a prototype *NOPASS parameter restriction.

The codepage must be indicated as a numeric decimal value as shown in this sample program.

close() — Close file

The close() function closes a file by using the file descriptor initialized by the open() function. This function also frees the descriptor field that can be reused by any future open() operations.

The following C syntax can be used as a reference to prototype the parameters required to call this function:

```
#include <unistd.h>
int close(int fildes);
```

Table 70 shows the parameters for this function. These parameters must be defined in the prototype used to call this API.

Table 70. Parameters for the close() function

Argument	Description	Use	RPG data type	C data type
	Return value	Output	Integer(10)	int
fildes	File descriptor	Input	Integer(10)	int

Here is the prototype used for the read API in an RPG IV format:

```
 * IFSPROTO from IFSSRC in RPGISCOOL (part 2 of 5)
 * close function prototype                                2
Dclose           PR           10I 0 Extproc('close')
D FileDesc                    10I 0 Value
```

The sample program CREATSTMF shows where we defined the variable used by the API, the return value field, and the call statement for closing the stream file (see marker 2, 3, and 18).

The return value returned by the API indicates if the operation was successful (0) or not (-1). In case of problems, the *errno* global variable is set to indicate the error. The *errno* value can be retrieve using various C APIs such as: perror(), __errno(), and strerror(). These APIs are explained later in this chapter.

CREATSTMF: A small sample program creating a stream file object

This sample program creates a stream file named "master" in a directory called "/rpgiscool" or clears the content of an already exiting one. The stream file created by this program is used as the master file in our later examples.

```
 * CREATSTMF from IFSSRC in RPGISCOOL
 ************************************************************
 * Program: CREATSTMF
 *
 * This program demonstrate the usage of the open() IFS APIs
 *  (UNIX-type) to create a stream file.
 * If the stream file already exist, its content will be cleared.
 * The program also close the file.
 *
 * **** Module needs to be bind with module DSPERROR
 *       during pgm creation ***
 *************************************************************
H option(*srcstmt) bnddir('QC2LE')

 * Prototypes required for the IFS APIs                        2
 /COPY RPGISCOOL/IFSSRC,IFSPROTO

 * Prototypes requires for DspError subprocedure
 /COPY RPGISCOOL/IFSSRC,ERRPROTO

 * open function return value definition                     4
D FileDesc        S             10I 0 Inz

 * Path name variables                                       5
D Path            S            100A   Inz('/rpgiscool/master')

 * Other variables for the IFS APIs used                      7
D Oflag           S             10I 0 Inz
D Mode            S             10I 0 Inz
D CodePage        C                   437

 * API return value fields                                    3
D RC              S             10I 0

 * Constants used by the program.
D Null            C                   X'00'

 * read API() definitions for oflag and mode parameters       4
 /COPY RPGISCOOL/IFSSRC,OPENDFN

 * Set file status flag for stream file creation where the file is
 *  open for both reading and writing, a codepage will be specified
 *  in the codepage parameter and if the file exist, its size will
 *  be truncated to zero (cleared)                              7
C                   Eval      oflag = O_CREATE + O_RDWR + O_CODEPAGE
C                                       + O_TRUNC
 * Set security attribute parameter                             7
C                   Eval      mode = S_IRWXU + S_IRWXG +
C                                       S_IROTH + S_IXOTH

 * Create stream file                                        7
C                   Eval      FileDesc = open(%trimr(path) : oflag :
C                                             mode : codepage)
C                   If        FileDesc = -1
C                   CallP     DspError('IFS open')
C                   Else

 * Close the file                                             18
C                   Eval      RC = close(Filedesc)
C                   If        RC = -1
C                   CallP     DspError('IFS close')
C                   EndIf
C                   EndIf

C                   Eval      *inLR = *on
```

Try it yourself

To recreate the previous program sample on your system, you can use the following commands:

```
CRTRPGMOD MODULE(rpgiscool/creatstmf) SRCFILE(rpgiscool/ifssrc)
CRTPGM PGM(rpgiscool/creatstmf) MODULE(rpgiscool/creatstmf
 rpgisccol/dsperror)
```

The program creates a stream file *master* in the directory */rpgiscool*. The directory must exit prior to calling the program. To create the directory in the IFS, you can use the command:

```
MD DIR('/rpgiscool')
```

Do not forget to create the DSPERROR module before, with the instructions provided in “The DspError subprocedure” on page 302. The three /COPY members used in this program must also exist where they are referenced. These members are defined in the following section.

Also, the IFS APIs and the C runtime functions are included in a binding directory named QC2LE. We chose to specify the binding directory (BDNDIR) on the H-Spec of our modules. Instead, the binding directory can be explicitly specified on the CRTPGM command.

5.7.4.2 Writing data to a stream file object

This section explains the usage of the IFS API used to write data to a stream file object. The write() -- Write to Descriptor IFS API is used. The two APIs described in the previous section are also used in this example.

The sample program WRITESTMF will help you understand the usage of those APIs to open a stream file, write text data into it, and then close the stream file. All those APIs are documented in *System API Reference OS/400 UNIX-Type APIs,* SC41-5875, in the chapter "Integrated File System APIs".

Note: None of the following APIs required the use of the path name structure described earlier.

In the following sections, you can correlate the marker of each API example to the main source code in 5.7.4.6, “Sample program code” on page 273.

write() — Write to Descriptor

The write() function writes a certain number of bytes from a memory area indicated by a buffer pointer to a file identified by its file descriptor. The data conversion from the stream file CCSID to the job or program run-time CCSID is handled by the API(), if the O_TEXTDATA status flag is specified on the open() instruction.

If the O_APPEND status flag was set on the file opening, the file offset is positioned at the end of the file, so the data can be appended after the existing data. Otherwise, the write function starts writing data at the beginning of the file. The file offset changes by the number of bytes written to the file.

The API returns to the number of bytes written in the return value parameter.

A value of -1 in the number of byte read indicates that the read was not successful. The *errno* value can be retrieved by using various C APIs such as: perror(), __errno(), and strerror(). These APIs are explained later in this chapter.

The following C syntax can be used as a reference to prototype the parameters required to call this function:

```
#include <unistd.h>
   ssize_t write
     (int file_descriptor, const void *buf, size_t nbyte);
```

Table 71 shows the parameters for this function. These parameters must be defined in the prototype used to call this API.

Table 71. Parameters for the write() function

Argument	Description	Use	RPG data type	C data type
	Return value	Output	Integer(10)	ssize_t
file_descriptor	File descriptor	Input	Integer(10)	int
buf	Buffer pointer	Input	Pointer	void *
nbyte	number of bytes to write	Input	Numeric Unsigned(10)	size_t

Here is the prototype used for the write API in an RPG IV format:

```
 * IFSPROTO from IFSSRC in RPGISCOOL (part 4 of 5)
 * write function prototype                                  2
D write           PR            10I 0 Extproc('write')
D  FileDescO                    10I 0 Value
D  bufferO@                       *   Value
D  nbyte                        10U 0 Value
```

The sample program WRITESTMF shows where we defined the parameters used by the API, the return value field, and the call statement (see marker 2, 3, and 16).

WRITESTMF: A small program to write data to a stream file

Here is an example of a simple program, which writes data to an already existing stream file. The stream file updated by this program will be used as the master file in our later examples.

An exercise for you

Notice that this program uses the commercial at (@) to indicate that the variable is a pointer. This breaks the style guide rule found in 2.1.3.3, “Avoid using special characters (for example, @, #, $) when naming items” on page 22.

Avoid the "@" symbol to indicate that the variable is a pointer. A good method to use is the "Hungarian Notation", where the first character of the variable name indicates the data type of the variable, for example, pCustNbr or cActStsCde.

```
 * WRITESTMF from IFSSRC in RPGISCOOL
 ***************************************************************************
 * Program: WRITESTMF
 *
 * This program demonstrate the usage of some IFS APIs (UNIX-type)
 *  to handle stream file. The program open an existing stream
 *  file which should be empty and writes 10 records in the stream
 *  file, then close the file
 *
 * **** Module needs to be bind with module DSPERROR during pgm creation ***
 ***************************************************************************
H option(*srcstmt) bnddir('QC2LE')

 * Prototypes required for the IFS APIs                            2
 /COPY RPGISCOOL/IFSSRC,IFSPROTO
 * Prototypes requires for DspError subprocedure
 /COPY RPGISCOOL/IFSSRC,ERRPROTO

 * Variables for the IFS APIs used                               3
D Path            S            100A   Inz('/rpgiscool/master')
D FileDesc        S             10I 0 Inz
D Oflag           S             10I 0 Inz
D CodePage        C                   437
D buffer          S            200A   Inz(*blank)
D  buffer@        S               *   Inz(%addr(buffer))

 * API return value fields                                       3
D RC              S             10I 0

 * read API() definitions for oflag and mode parameters            4
 /COPY RPGISCOOL/IFSSRC,OPENDFN

 *  The data in the following DS will be written as a series of records
 *    to the IFS file that we open
D MyData          DS
D                              198A   Inz('Hello &name&,')
D                              198A   Inz(*blanks)
D                              198A   Inz('    Here is an email -
D                                     sent by &lng& program.')
D                              198A   Inz(*blanks)
D                              198A   Inz('    Please accept -
D                                     my best wishes for your -
D                                     futur success using')
D                              198A   Inz('    this cool language')
D                              198A   Inz(*blanks)
D                              198A   Inz(*blanks)
D                              198A   Inz('Regards,')
D                              198A   Inz('The RPGisCool team.')
D RecordData                   198A   Dim(10) Overlay(MyData)

 * Stand-alone variables
D Count           S              5P 0 Inz
 * Constants used by the program.
D CRLF            C                   X'0D25'
D Null            C                   X'00'

 * Set file status flag for stream file open where the file is open
 *  for write only and data conversion will occurs (ASCII to EBCDIC)
 *  autmatically when the file data will be written              6

C                   Eval      oflag = O_WRONLY + O_TEXTDATA

 * Openning the file, receiving a file descriptor                 6
C                   Eval      FileDesc = open(%trimr(path) : oflag)
C                   If        FileDesc = -1
C                   CallP     DspError('IFS open')
C                   Else

 * For every record in the array
C                   For       Count = 1 to %Elem(RecordData)
C                   Eval      buffer = %trimr(RecordData(Count))
C                                         + CRLF
 *  Write records to the file                                    16
C                   Eval      rc = write(FileDesc : buffer@ :
C                                         %len(%trimr(buffer)))
C                   If        RC = -1
C                   CallP     DspError('IFS write')
C                   EndIf
```

```
C                   EndFor

 * Close the file                                              18
C                   Eval      RC = close(FileDesc)
C                   If        RC = -1
C                   CallP     DspError('IFS close')
C                   EndIf
C                   EndIf

C                   Eval      *inLR = *on
```

Try it yourself

To recreate the previous program sample on your system, you can use the following commands:

```
CRTRPGMOD MODULE(rpgiscool/writestmf) SRCFILE(rpgiscool/ifssrc)
CRTPGM PGM(rpgiscool/writestmf) MODULE(rpgiscool/writestmf
 rpgisccol/dsperror)
```

The program writes to a stream file named *master* in the directory */rpgiscool*, which you can create using the CREATSTMF sample program described previously. Since the directory and the stream file must exits prior to calling the program, you can use the following command to create the directory and the stream file in the IFS, if you prefer not to use the CREATSTMF program:

```
MD DIR('/rpgiscool')
EDTF STMF(/'rpgiscool/master')
 (and exit the EDTF editor by pressing F3 twice to save the file.)
```

Note: The EDTF utility is only available at V4R4 and beyond. If you are running at V4R3 or prior, you can create an empty stream file by using the following commands:

```
CRTPF PF(rpgiscool/master) RCDLEN(1)
CPYTOSTMF FROMMBR('/qsys.lib/rpgiscool.lib/master.file/master.mbr')
 TOSTMF('/rpgiscool/master')
DLTF FILE(rpgiscool/master)
```

Do not forget to create the DSPERROR module before, with the instructions provided in “The DspError subprocedure” on page 302. The three /COPY members used in this program must also exist where they are referenced. These members are defined in the following section.

Also, the IFS APIs and the C runtime functions are included in a binding directory named QC2LE. We chose to specify the binding directory (BDNDIR) on the H-Spec of our modules. Instead, the binding directory can be explicitly specified on the CRTPGM command.

5.7.4.3 Reading stream file content and changing the file offset position

This section explains the usage of the last two IFS API used in the main sample application MERGESTMF. Using these two APIs, you can read the content of a stream file and change the file offset (cursor) position.

The list of IFS APIs used are:

- read() — Read from Descriptor
- lseek() — Set File Read/Write Offset

We provide a sample coding abstract to use each of them. The main sample program MERGESTMF will guide you to a more practical example.

All of these APIs are documented in *System API Reference OS/400 UNIX-Type APIs,* SC41-5875, in the chapter "Integrated File System APIs".

Note: None of the following APIs require the use of the path name structure described earlier.

In the following sections, you can correlate the marker of each API example to the main source code in 5.7.4.6, “Sample program code” on page 273.

read() — Read from descriptor

The read() function reads a certain number of bytes of a file referred by a file descriptor into a memory area indicated by a buffer pointer. The data conversion from the stream file CCSID to the job or program run-time CCSID is handled by the API() if the O_TEXTDATA status flag has been specified on the open() instruction.

The following C syntax can be used as reference to prototype the parameters required to call this function:

```
#include <unistd.h>
   ssize_t read(int file_descriptor,
                void *buf, size_t nbyte);
```

Table 72 shows the parameters for this function. These parameters must be defined in the prototype used to call this API.

Table 72. Parameters for the read() function

Argument	Description	Use	RPG data type	C data type
	Return value	Output	Integer(10)	ssize_t
file_descrip tor	File descriptor	Input	Integer(10)	int
buf	Buffer pointer	Output	Pointer	void *
nbyte	number of bytes to be read	Input	Numeric Unsigned(10)	size_t

Here is the prototype used for the read API in an RPG IV format:

```
 * IFSPROTO from IFSSRC in RPGISCOOL (part 3 of 5)
 * read function prototype                                      2
D read            PR            10I 0 Extproc('read')
D  FileDescI                    10I 0 Value
D  bufferI@                       *   Value
D  nbyte                        10U 0 Value
```

Here is an abstract of our sample program where we defined the variable used by the API, the return value field and the call statement:

```
 * Prototypes required for the IFS APIs                         2
 /COPY RPGISCOOL/IFSSRC,IFSPROTO
...
 * Variables for read() APIs                                    3
D bufferI         S            200A   Inz(*blank)
D  bufferI@       S              *    Inz(%addr(bufferI))
D nbyteread       S             10I 0 Inz(0)
D nbyteset        S             10U 0 Inz(200)
```

```
...
 * Read() structure definition                                      4
 /COPY RPGISCOOL/IFSSRC,OPENDFN
...
 * read stream file (filedescI) for 200 bytes and place results in
 *    bufferI                                                        11
C                   Eval      nbyteread = read(filedescI : bufferI@ :
C                                                nbyteset)
```

The number of bytes to be read (nbyteset) indicates how many bytes to read into the stream file identified by its file descriptor. The bytes read are placed in a memory location represented by the buffer pointer passed to the API. The API initializes this location with the data read from the file and returns to number of bytes read in the return value.

A value of -1 in the number of byte read indicates that the read was *not* successful. The *errno* value can be retrieved by using various C APIs such as: perror(), __errno(), and strerror(). These are explained later in this chapter.

The read() operation begins reading the file at the file offset associated with the file descriptor. On a successful read, the file offset is changed by the number of bytes read. This file offset can be manually changed by using the lseek() API, which is described in the following section.

lseek() — Set file read/write offset

The lseek() function changes the current file offset to a new position in a file referred by its file descriptor. The new position is the given byte offset from the starting position specified by the *whence* parameter. After using lseek() to position the cursor offset to a new location, the next I/O operation on the file, like read() or write(), begins at that new location.

The following C syntax can be used as reference to prototype the parameters required to call this function:

```
#include <unistd.h>
   off_t lseek(int file_descriptor, off_t offset, int whence)
```

Table 73 shows the parameters for this function. These parameters must be defined in the prototype used to call this API.

Table 73. Parameters for the write() function

Argument	Description	Use	RPG data type	C data type
	Return value	Output	Integer(10)	off_t
file_descrip tor	File descriptor	Input	Integer(10)	int
offset	offset	Input	Integer(10)	int
whence	the starting position	Input	Integer(10)	int

Here is the prototype used for the lseek API in an RPG IV format:

```
 * IFSPROTO from IFSSRC in RPGISCOOL (part 5 of 5)
 * lseek function prototype                                    2
D lseek           PR            10I 0 Extproc('lseek')
D  FileDesc                     10I 0 Value
D  offset                       10I 0 Value
D  whence                       10I 0 Value
```

Here is an abstract of our sample program where we defined the variable used by the API, the return value field, and the call statement:

```
 * Prototypes required for the IFS APIs                         2
 /COPY RPGISCOOL/IFSSRC,IFSPROTO
...
 * variables for lseek API                                      3
D nbyteread       S             10I 0 Inz(0)
D whence          S             10I 0 Inz(SEEK_CUR)
...
 * Lseek() structure definition                                 4
 /COPY RPGISCOOL/IFSSRC.LSEEKDFN
...
 * reposition to file offset before the previous read           15
C                   Eval      rc = lseek(FileDescI :
C                                        nbyteread:
C                                        whence)
```

In this example, the *nbyteread* value was initialized by the previous read() instruction. The lseek function positions the offset to its original position before the read occurs. Perhaps the file can be read again with a larger buffer for example. The offset is moved from the current file offset (whence) minus the number of bytes read previously.

The *whence* parameter represents the starting position of the offset movement. The API documentation describes the value of this parameter as:

SEEK_SET	The start of the file
SEEK_CUR	The current file offset in the file
SEEL_END	The end of the file

The C definition of each value available for this parameter can be found in the member UNISTD of the source file QSYSINC/H from the system openness include licensed program. The following file sample includes the C definition of the whence values as comments with the equivalent definition of those in an RPG IV format:

```
 filename: RPGISCOOL/IFSSRC(LSEEKDFN)
 * structure definition for lseek() function                    4
 *    from QSYSINC/H.UNISTD member

 */*****************************************************************/
 */*   Constants for lseek()                                      */
 */*****************************************************************/
 *   #define SEEK_SET    0        /* Seek to given position        */
D SEEK_SET        C                 0
 *   #define SEEK_CUR    1        /* Seek relative to current
 *                                   position                      */
D SEEK_CUR        C                 1
 *   #define SEEK_END    2        /* Seek relative to end of file  */
D SEEK_END        C                 2
```

We recommend that you /COPY this member file into your RPG program for ease of use and comprehension, as shown in the previous example.

5.7.4.4 Overall concept of the sample application

The sample application used in this section merges text passed as a parameter to a master file, where the variable text location is defined. The result of the merging produces an output stream file. As example of using this master file is shown here:

```
stream file name: /rpgiscool/master:
************Beginning of data**************
Hello &name&,

    Here is an email sent by &lng& program.

    Please accept my best wishes for your future success using
    this cool language.

Regards,
The RPGisCool team.
************End of Data*******************
```

Note: A carriage return and line feed character (x'0d25') end each line.

You can use the following command to pass the required parameters to the RPG IV program. Using the master file above, this example creates a stream file named */rpgiscool/out*, where the two variables used, &var1& and &lng&, are replaced by the corresponding values passed on the Merge Stream File (MERGESTMF) command:

```
MERGESTMF MASTER('/rpgiscool/master') OUTPUTFILE('/rpgiscool/out')
ELEMENTS(('&name&' 'World') ('&lng&' 'an RPG IV'))
```

In this example, we decided to use a command object because the OS/400 command processing has technical difficulties handling character strings longer than 32 bytes as parameters. For more information, see the chapter "Passing Parameters between Programs and Procedures" in *CL Programming,* SC41-5721.

Here is the content of the result file:

```
stream file name: /rpgiscool/out:
************Beginning of data**************
Hello World,

    Here is an email sent by an RPG IV program.

    Please accept my best wishes for your future success using
    this cool language.

Regards,
The RPGisCool team.
 ************End of Data*******************
```

Stream file display and edit

The Edit File (EDTF) and Display Stream File (DSPSTMF) commands are available through a PTF. See the label box "Displaying and editing stream files" on page 253.

Go directly to 5.7.4.5, "Anatomy of the MERGESTMF sample program" on page 271, for a detailed explanation of the sample program.

5.7.4.5 Anatomy of the MERGESTMF sample program

As described in 5.7.4.4, "Overall concept of the sample application" on page 270, this sample program merges data passed as a parameter with data already specified in a master stream file to produce a new stream file.

For the benefits of providing a simple example, this program can replace variables by their associated value only if the variable identifier is fully contained within a 200 byte or less record length. We know there is better way to do this, but we intended to keep this IFS API sample program as simple as possible.

Program MERGESTMF outline

Here is the outline of the Merge Stream file (MERGESTMF) program:

- **Field definitions**

  1 Declare the prototype and the procedure interface for the program input parameters.

2 Declare the prototype used to call the IFS APIs and the DSPerror external subprocedure.

3 Define the variables used to pass or receive data from the IFS APIs.

4 Define the structure definition used by the IFS APIs.

5 This section also defines various standalone global variables used by this program.

- **File opening**

6 Open the master file in read only mode and enable text data conversion. In case of an error, call the DspError external subprocedure to display the error, and skip the rest of the process.

7 Open the output file for creation, in read/write mode with codepage specified and data truncation if the file already exists. The owner and the primary group member have *RWX authorities and *PUBLIC has *RX authority. In case of an error, call the DspError external subprocedure to display the error, and skip the rest of the process.

8 Close the output file. In case of an error, call the DspError external subprocedure to display the error, and skip the rest of the process.

9 Reopen the output file, this time, specify the write only mode and the text data conversion function. In case of an error, call the DspError external subprocedure to display the error, and skip the rest of the process.

- **Main process**

10 The Main process is executed if no errors are encountered during the file opening section. The program loops until there are no more bytes to read in the master file or an error occurred with any of the IFS APIs.

11 The read instruction reads 200 bytes of the master file to the input buffer string. In case of an error, call the DspError external subprocedure to display the error, and skip the rest of the process (see marker 17).

12 If no errors occur during the read operation and the number of bytes returned is greater than zero, the input buffer string is scanned to look for the carriage return/Line feed characters (CRLF) represented by the characters x'0d25'. If found, the output buffer string is initialized with the data from the beginning of the input buffer string until (and including) the CRLF character.

13 For every merge variable specified as an input parameter, the variable identifier and the replacement value are extracted from their input parameter. The output buffer string is scanned with the variable identifier and replaced by the replacement value if found in the record. See "MERGESTMF program notes" on page 277 for more information.

14 The output buffer string is written to the output file, using the number of bytes of the output buffer string from the beginning until the CRLF is encountered for the length of the string to be written to the stream file. In case of an error, call the DspError external subprocedure to display the error, and skip the rest of the process. See "MERGESTMF program notes" on page 277 for more information.

15 The file offset is changed back to the byte after the CRLF character has been encountered in the input buffer string, so the next read can start from this offset. In case of an error, call the DspError external subprocedure to

display the error, and skip the rest of the process. See "MERGESTMF program notes" on page 277 for more information.

16 If no CRLF characters were found in step 12, the input buffer string is written to the output file using the number of bytes read as the length of the data being written to the stream file. In case of an error, call the DspError external subprocedure to display the error, and skip the rest of the process.

- **File closing**

18 Both stream files are closed. In case of an error, call the DspError external subprocedure to display the error, and skip the rest of the process.

Why use two opens when creating a new file?

There is an important consideration that you need to take care of when you are creating and then reading or writing to a stream file in the same program. For the EBCDIC to ASCII (or the opposite) translation to occur between your application and the ASCII stream file, you must open the file twice, in this sequence:

1. Open the file for creation with the file status flag set to O_CREAT, the proper mode, and the codepage.
2. Close the file.
3. Re-open the file with the proper file status flag, so the EBCDIC to ASCII translation can be handled.

When the open function is used to create a new file, the codepage parameter indicates under which codepage this file is created. It is usually an ASCII codepage, such as 437, since stream files are mainly ASCII files. On file creation, the codepage also indicates the character set of the data expected to be passed to the write() API. The codepage specified was an ASCII one, and as your RPG IV program is working under your job CCSID, which is an EBCDIC one (unless specify otherwise). Because of this, the data would needs to be converted manually before it is passed to IFS APIs using this file. No automatic conversion occurs even if the O_TEXTDATA status flag is specified.

This is the reason why a second open is needed. When the open function is performed on an already existing file, and the O_TEXTDATA status flag is specified, the data read or write is converted automatically between this code page and the file codepage. The recommended way is to leave the default one (job CCSID) and to specify the O_TEXTDATA status flag.

This logic may appear confusing, but it works. You can find more information on this subject in the open() API documentation, in *Using the codepage parameter* section of the *System API Reference OS/400 UNIX-Type APIs*, SC41-5875.

5.7.4.6 Sample program code

In this section, you find the complete sample code used in the MERGESTMF sample program. This sample program is made of two modules, a user created command and multiple /COPY members, which are described as follows:

- The main module (MERGESTMF)
- The User created command object
- The Main module prototype for input parameter (/COPY member)
- The IFS APIs prototypes (/COPY member)
- The structure definitions used by the open() API (/COPY member)
- The structure definition used by the lseek() API (/COPY member)
- The DspError module (DSPERROR)
- The DspError subprocedure prototype (/COPY member)

This program can use the master file created with the small sample programs found in “CREATSTMF: A small sample program creating a stream file object” on page 262 and “WRITESTMF: A small program to write data to a stream file” on page 265.

The main module MERGESTMF

Here is the code of the main module. It includes markers that relate the RPG IV code to the explanations in “MERGESTMF program notes” on page 277.

An exercise for you

Notice that this program uses the commercial at (@) to indicate that the variable is a pointer. This breaks the style guide rule found in 2.1.3.3, “Avoid using special characters (for example, @, #, $) when naming items” on page 22.

Avoid 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, for example, pCustNbr or cActStsCde.

```
 * MERGESTMF from IFSSRC in RPGISCOOL
 ***************************************************************************
 *MERGESTMF
 *
 * This program demonstrate the usage of some IFS APIs (UNIX-type)
 *  to handle stream file. The program reads a master stream file, converts
 *  a user variable coded in the text to a value pass in parameters, and
 *  write the new string to another stream file.
 *
 * The following Unix type API's are used in this example:
 *  Open(), read(), write(), close(), lseek(),
 * Also, this program use 2 C-runtime functions to display the any errors
 *  encounters while using the UNIX APIs: __errno(), strerror()
 *
 * This program receives an 3 input parameters: pathname for the master file,
 *   string to replace the variable "&var1&" and the output file path name.
 *
 * **** Module needs to be bind with module DSPERROR during pgm creation ***
 ***************************************************************************

H option(*srcstmt) bnddir('QC2LE')

 * Program prototypes
 /COPY RPGISCOOL/IFSSRC,MERGEPROT

 * Input parameters                                                     1
D Mergestmf       PI
D  inputpath                   100A
D  outputpath                  100A
D  elemstr                    1142A


 * Prototypes required for the IFS APIs                                       2
 /COPY RPGISCOOL/IFSSRC,IFSPROTO
 * Prototypes requires for DspError subprocedure
 /COPY RPGISCOOL/IFSSRC,ERRPROTO

 * Variables for Open() API                                          3
D oflag           S             10I 0 Inz(0)
D mode            S             10U 0 Inz(0)
D codepage        S             10U 0 Inz(437)
D Rc              S             10I 0 Inz(0)
 * Variables for Read(), Write() and Close() APIs
D FileDescI       S             10I 0 Inz(0)
D FileDescO       S             10I 0 Inz(0)
D bufferI         S            200A   Inz(*blank)
D  bufferI@       S               *   Inz(%addr(bufferI))
D bufferO         S            300A   Inz(*blank)
D  bufferO@       S               *   Inz(%addr(bufferO))
D nbyteset        S             10U 0 Inz(200)
D nbyteread       S             10I 0 Inz(0)
 * Variables for lseek() API
D whence          S             10I 0 Inz(SEEK_CUR)

 * read API() definitions for oflag and mode parameters               4
 /COPY RPGISCOOL/IFSSRC,OPENDFN
 * lseek API() definitions for whence parameter
 /COPY RPGISCOOL/IFSSRC,LSEEKDFN

 * stand-alone variables                                       5
D Null            S              1A   Inz(x'00')
D Pos             S              5U 0 Inz(0)

D CRLFPos         S             10I 0 Inz(0)
D CRLF            S              2A   Inz(x'0d25')

 * Variables required for the list within list parameter (elemstr)           5
D pElemstruct     S               *
D pElems          S               *

D elemstruct      DS                  based(pElemstruct)
D  numelems                      5I 0
D  displacements                 5I 0 dim(10)

D elems           DS                  based(pElems)
D  num                           5I 0
D  var                          10A
D  txt                         100A

D elemidx         S             10I 0 Inz(0)
D elemvar         S             10A   varying dim(10)
D elemtxt         S            100A   varying dim(10)


 * Set file status flag for stream file read
C                   Eval      oflag = O_RDONLY + O_TEXTDATA
 * Open stream file for read                                                  6
C                   Eval      FileDescI = open(%trimr(inputpath) :
C                                               oflag)
C                   If        FileDescI = -1
C                   CallP     DspError('OpenIn')
C                   Else

 * Set file status flag for stream file creation
C                   Eval      oflag = O_CREATE + O_RDWR + O_CODEPAGE
C                                       + O_TRUNC
C                   Eval      mode = S_IRWXU + S_IRWXG +
C                                       S_IROTH + S_IXOTH
 * Create stream file                                                 7
C                   Eval      FileDescO = open(%trimr(outputpath) :
C                                           oflag : mode : codepage)
C                   If        FileDescO = -1
C                   CallP     DspError('OpenOut1')
C                   Else
 * If created, Close the file                                        8
C                   Eval      rc = close(FileDescO)
C                   If        rc = -1
C                   CallP     DspError('CloseOut1')
C                   Else

 * If created and close, Open stream file for write                  9
C                  Eval      oflag = O_WRONLY + O_TEXTDATA
C                  Eval      FileDescO = open(%trimr(outputpath) :
```

```
C                                           oflag)
C                   If        FileDescO = -1
C                   CallP     DspError('OpenOut2')
C                   Endif
C                   Endif
C                   Endif
C                   Endif

 * Do Main process until no more data or error                  10
C                  If        FileDescI <> -1 and FileDescO <> -1

 * Do Read the file until no more data or error
C                   DoU       nbyteread <= 0 or
C                              FileDescI = -1 or rc = -1

 * Read the Input File                                          11
C                   Eval      nbyteread = read(FileDescI : bufferI@ :
C                                                nbyteset)

C                   If        nbyteread > 0 and FileDescI >= 0

 * Scan buffer for the first "carriage return" and "line feed"
 *  character                                                   12
C                   Eval      CRLFpos = %scan(CRLF : bufferI)

 * If CRLF found, Set Output buffer with data including first CRLF
C                   If        CRLFpos > 0
C                   Eval      bufferO = %subst(bufferI :
C                                          1 : CRLFpos + 1)

 * pre-process the list within list parameter (elemstr) using basing pointer   13
C                   Eval      pElemstruct = %addr(elemstr)
C                   For       elemidx = 1 to numelems
C                   Eval      pElems = %addr(elemstr) +
C                                         displacements(elemidx)
C                   Eval      elemvar(elemidx) = %trim(var)
C                   Eval      elemtxt(elemidx) = %trim(txt)
C                   Endfor

 * For every elements specified as parameters, scan for the variable, if
 * found, replace the variable by it's associated text.
C                   For       elemidx = numelems downto 1

C                   Eval      pos = %scan(elemvar(elemidx) : bufferO)
C                   If        pos > 0
C                   Eval      bufferO = %replace(elemtxt(elemidx) :
C                                                 bufferO : pos :
C                                                 %len(elemvar(elemidx)))
C                   Endif
C                   Endfor

 * Write Output buffer to Output file                           14
C                   Eval      rc = write(FileDescO : bufferO@ :
C                                   %scan(CRLF : bufferO) + 1)
 * If error occured during write
C                   If        rc = -1
C                   CallP     DspError('Write1')
C                   Endif

 * Move back file offset to character after the first CRLF in the file
 *  and use lseek to position cursor so read() can read at the beginning
 *  of the next line                                            15
C                   Eval      rc = lseek(FileDescI :
C                                    - ((nbyteread - CRLFpos) - 1) :
C                                       whence)
C                   If        rc = -1
C                   CallP     DspError('lseekIn')
C                   Endif

 * String with no CRLF found, write complete string             16
 *  to output file
C                   Else
C                   Eval      rc = write(FileDescO : bufferI@ :
C                                    nbyteread)

 * If error occured during write
C                   If        rc = -1
C                   CallP     DspError('Write2')
```

```
C                   Endif
 * Endif CRLF found
C                   Endif

 * If error occured during read                                        17
C                   Else
C                   If        FileDescI = -1
C                   CallP     DspError('ReadIn')
C                   Endif
C                   Endif

C                   Enddo

 * Close the files                                                     18
C                   Eval      rc = close(FileDescI)
C                   If        rc = -1
C                   CallP     DspError('CloseIn')
C                   Endif

C                   Eval      rc = close(FileDescO)
C                   If        rc = -1
C                   CallP     DspError('CloseOut2')
C                   Endif

 * Endif both opens successfull
C                   Endif

 * End of program
C                   Eval      *inLR = *on
```

MERGESTMF program notes

1 To add more flexibility to this sample program, we allowed 10 different variable names and values to be passed to the program as input parameters. The maximum length of a variable name is 10 character. Its replacing value can be up to 100 characters. We defined them in the command object as a list within a list parameter type. You can find more information on this definition in *CL Programming*, SC41-5721, in Section 9.4.3 "Defining Lists within Lists."

In the Calculation Specification, the markers **5** and **13** indicate where information is retrieved out of this list parameter.

13 Please refer to section **1** and Section 9.4.3 "Defining Lists within Lists", in the manual *CL Programming*, SC41-5721, for more information on the values passed in this list parameter. In this program example, we use based data structures and address concatenation (with the displacement) to find the correct value in the list parameter.

The %replace function, new at V4R4, replaces the variable name by its associated value and removes the leading and trailing blanks in the replacement value. With the %replace function, the %scan function is used to find the starting position of where the insertion in the existing text needs to be done. The %len function is used to determine the length of the text to be inserted.

14 The length of the data written to the file is determined by the &SCAN function on the CRLF character. Since this is a two-byte character and the %SCAN reports the location of the first byte, one is added to the location to indicate the length of the string.

15 Because the last successful read() operation set the cursor location (offset) of the Input File after the last byte read, we use the number of bytes read minus the position of the CRLF, minus 1, since it is a two-byte indicator (as described above), to subtract the offset from its current location.

The user created command object MERGESTMFC

The filename used for the user created command object is MERGESTMFC. Here is the sample code:

```
/* filename: RPGISCOOL/IFSSRC(MERGESTMFC)
/* Command definition for MERGESTMF program */
           CMD        PROMPT('Merge Stream File')
           PARM       KWD(MASTER) TYPE(*PNAME) LEN(100) +
                        PROMPT('Master file name' 1)
           PARM       KWD(OUTPUTFILE) TYPE(*PNAME) LEN(100) +
                        PROMPT('Output file name' 2)
           PARM       KWD(ELEMENTS) TYPE(ELEM) MAX(10) PGM(*YES) +
                        PROMPT('Merge Elements definition' 3)
ELEM:      ELEM       TYPE(*CHAR) LEN(10) EXPR(*YES) CASE(*MIXED) +
                        PROMPT('Variable name')
           ELEM       TYPE(*CHAR) LEN(100) MIN(1) CASE(*MIXED) +
                        EXPR(*YES) PROMPT('Replacing value')
```

Creating the command

You can create the command object named MERGESTMF with the following command:

```
CRTCMD CMD(RPGISCOOL/MERGESTMF) PGM(MERGESTMF) SRCFILE(RPGISCOOL/IFSSRC)
SRCMBR(MERGESTMFC)
```

This command implies that the program created will be named MERGESTMF.

The main module prototype MERGEPROT for input parameter

The filename for the main module prototype is MERGEPROT. Here is the program prototype defining the input parameters interface, which is included in the main source with the /COPY instruction:

```
 * filename: RPGISCOOL/IFSSRC(MERGEPROT)
 * MergeStmf prototypes                                      1
D MergeStmf       PR                  Extpgm('MERGESTMF')
D  Inputpath                   100A
D  Outputpath                  100A
D  mergestr                   1142A
```

The IFS APIs prototype IFSPROTO

Here are the definitions of the prototypes required to call the IFS APIs used in this program. The member name is IFSPROTO. Refer to the definition of each of them in 5.7.4.3, “Reading stream file content and changing the file offset position” on page 267.

```
 * IFSPROTO from IFSSRC in RPGISCOOL
 * Prototypes for IFS APIs                                       2
 * Open() - Open File API
D open            PR            10I 0 ExtProc('open')
D  path@                          *   Value Options(*string)
D  oflag                        10I 0 Value
D  mode                         10U 0 Value Options(*nopass)
D  codepage                     10U 0 Value Options(*nopass)

 * read() -- Read from Descriptor
D read            PR            10I 0 Extproc('read')
D  FileDescI                    10I 0 Value
D  bufferI@                       *   Value
D  nbyte                        10U 0 Value

 * write() -- Write to Descriptor
D write           PR            10I 0 Extproc('write')
D  FileDescO                    10I 0 Value
D  bufferO@                       *   Value
```

```
D  nbyte                        10U 0 Value

 * close() -- Close File or Socket Descriptor
D close           PR            10I 0 Extproc('close')
D  FileDesc                     10I 0 Value

 * lseek()--Set File Read/Write Offset
D lseek           PR            10I 0 Extproc('lseek')
D  FileDesc                     10I 0 Value
D  offset                       10I 0 Value
D  whence                       10I 0 Value
```

The structure definitions used by the open() API

The member name used for the structure definition of the open API is *OPENDFN*. It should contain the definitions of the file access mode, status flags, and share mode, described in "Open file parameter: File access mode, status flag, and share mode" on page 258, with the security attributes described in "Open file parameter: Security attributes or permission bits (optional)" on page 259.

The structure definition used by the lseek() API

The member name used for the structure definition of the lseek API is *LSEEKDFN*. The contents of this member are described in "lseek() — Set file read/write offset" on page 269.

The DspError subprocedure prototype

The member name for the DspError subprocedure prototype is *ERRPROTO*. The content of this member is described in "Prototype for the DspError subprocedure" on page 301.

The DspError module (DSPERROR)

The DSPError module is used to display error messages that occur during the API processing. The subprocedure retrieves the content of the errno variable and retrieves the associated text. Both information, plus a text location input parameter is displayed on the screen. This external module needs to be bound at program creation time with the main module. There is more information on the DspError subprocedure in 5.7.6.2, "The DspError subprocedure" on page 301.

Try it yourself

To re-create the previous program sample on your system, you can use the following commands:

```
CRTRPGMOD MODULE(MERGESTMF) SRCFILE(IFSSRC)
CRTPGM PGM(MERGESTMF) MODULE(MERGESTMF DSPERROR)
```

Do not forget to create the DSPERROR module before. The instructions are provided in “The DspError subprocedure” on page 302.

Also, the IFS APIs and the C runtime functions are included in a binding directory named QC2LE. We chose to specify the binding directory (BDNDIR) on the H-Spec of our modules. Instead, the binding directory can be explicitly specified on the CRTPGM command.

For your information, the two programs “CREATSTMF: A small sample program creating a stream file object” on page 262 and “WRITESTMF: A small program to write data to a stream file” on page 265 are used to create and update the master stream file used by the MERGESTMF program in this section.

You can use the user created command, as described in 5.7.4.4, “Overall concept of the sample application” on page 270, to call the program.

5.7.5 Using more complex IFS APIs: Qp0lProcessSubtree()

This section demonstrates the usage of another IFS API named Qp0lProcessSubtree(). The primary function of this API is to search the directory under a specific path name. It selects and passes objects, one at a time, to an exit program that is identified on its call. The exit program can be either a procedure or a program.

5.7.5.1 Overall concept of the sample program SWEEP

This RPG IV program is based on the C sample program provided in Section 2.57.1, "Qp0lProcessSubtree()--Process a Path Name - Scenario 4," of *System API Reference OS/400 UNIX-Type APIs*, SC41-5875. This program processes a directory and all files and subdirectory within this directory, starting at a specific path name specified as one of the input parameters.

A user created command object is used to passed the parameters to the APIs. Since the parameters of the command are the same as the ones required by the API, with exception of the exit program pointer, they are explained along with the API parameters explanations. For the example, we are using the directory structure defined in Figure 33 on page 281, where a, b, c, d, e, and f are directories under the root file system, and t, u, v, w, x, y, z are stream files.

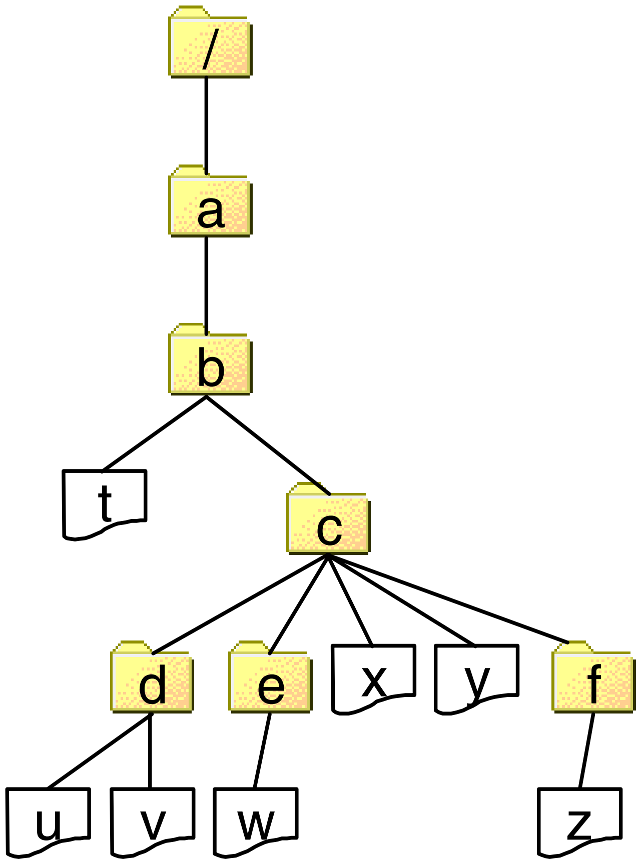


Figure 33. IFS structure sample with directories and stream files

This command can be used to call the sample program:

```
SWEEP PATHNAME('/a') OBJTYPES(*STMF) INEXCLUTY(*EXCL)
 USERTEXT('The stream file found by Sweep is:')
 INEXCLUPN('/a/b/c/d' '/a/b/c/e')
```

The following results can be obtain from the spooled file printed by the sample program.

```
The stream file found by Sweep is:
/a/b/c/f/z
The stream file found by Sweep is:
/a/b/c/x
The stream file found by Sweep is:
/a/b/c/y
The stream file found by Sweep is:
/a/b/t
Qp0lProcessSubtree() Successful
```

For each object or directory encountered during the process, the API calls an exit program, which prints the first 132 bytes of the path name (sample program limitation), along with User Text specified on the command. The API also provides the functionality of including or excluding the directories from the search or to search only specific object types. In this case, we excluded two directories and their associated stream files (/a/b/c/d and /a/b/c/e), and we only asked for the stream file (*STMF) object type. The exit program used by the API is a subprocedure defined in a different module as the main procedure calling the API. The exit program handles a path name in Unicode (UCS-2) format.

For this sample program to stay as simple as possible, the input parameters path name must not be greater than 100 bytes. We also assumed none of the path name process by the exit program would be larger than 8000 bytes. The command object is not necessary to use this API in an RPG program, but it provides a much simpler interface when passing the parameters from the command line.

5.7.5.2 The Qp0lProcessSubtree() API

The Qp0lProcessSubtree() API is a new IFS system function. The integrated file system is a part of OS/400 that supports stream input/output and storage management similar to a personal computer and UNIX operating systems, while providing an integrated structure over all objects stored in the AS/400. One of several features that allows this support to be accomplished is a hierarchical directory structure that allows objects to be organized like fruit on different branches of a tree. The Qp0lProcessSubtree() API is the first AS/400 system function to allow these directories or the objects within these directories to be selectively identified, collected, deleted, or otherwise processed, within or across physical file systems. It is a C integrated language environment (ILE) function that can be used in ILE C, ILE RPG, or ILE COBOL C/400 programs.

The Qp0lProcessSubtree() API starts at a caller-specified directory or path name and sweeps through all objects in the tree under that directory. The API invokes a caller-specified exit program for each object encountered that meets the input specifications from the caller. This exit program is either a procedure or a program and is designed by the API caller to complete the necessary processing of a selected object.

The Qp0lProcessSubtree() API is documented in the *System API Reference OS/400 UNIX-Type APIs,* SC41-5875, in the chapter "Integrated File System APIs".

The following C syntax can be used as a reference to prototype parameters required to call this function:

```
#include <Qp0lstdi.h>
  int Qp0lProcessSubtree (
    Qlg_Path_Name_T          *Path_Name,
    uint                      Subtree_level,
    Qp0l_Objtypes_List_t     *Objtypes_array_ptr,
    uint                      Local_remote_obj,
    Qp0l_IN_EXclusion_List_t *IN_EXclusion_ptr,
    uint                      Err_recovery_action,
    Qp0l_User_Function_t     *UserFunction_ptr,
    void                     *Function_CtlBlk_ptr, ...);
```

Table 74 on page 283 shows the parameters for this function. These parameters must be defined in the prototype used to call this API.

Table 74. Parameters for the Qp0lProcessSubtree() API

Argument	Description	Use	RPG data type	C data type
	Return value	Output	Integer(10)	int
Path_Name	Path Name	Input	*	Qlg_Path_Name_T
Subtree_level	Subtree_level	Input	Unsigned Numeric(10)	uint
Objtypes_array_ptr	Array of Object types structure	Input	*	Qp0l_Objtypes_List_t
Local_remote_obj	Local or Remote objects	Input	Unsigned numeric (10)	uint
In_Exclusion_ptr	Array of Includes or Excludes directories	Input	*	Qp0l_IN_EXclusion_List_t
Err_recovery_action	Handle action for error recovery	Input	Unsigned numeric (10)	unint
UserFunction_ptr	Pointer to the process a path name exit program (subprocedure or program)	Input	*	Qp0l_User_Function_t
Function_CtlBlk_ptr	Pointer to the function control block	Input	*	void *

Here is the API prototype in an RPG IV format:

```
 * PRCSUBTRPR from SWEEPSRC in RPGISCOOL (part 1 of 2)
 * Prototype for Qp0lProcessSubtree() API                          2
 *  From the System API Reference OS/400 UNIX-Type APIs
D ProcessSubtree   PR          10I 0 Extproc('Qp0lProcessSubtree')
D  PathName@                     *   Value options(*string)
 *                             Path_Name, see DS QLGPN
D  SubtreeLevel                10U 0 Value
 *                             Subtreee_level, 0=Yes, 1=No
D  ObjtypeArray@                 *   Value
 *                             Objtypes_array_ptr
D  LocalRemote                 10U 0 Value
 *                             Local_remote_obj, 0=Local+Remote,
 *                                            1=Local , 2=Remote
D  InExclusion@                  *   Value
 *                            In_Exclustion_ptr, see DS QP0INEXL
D  ErrRecovery                 10U 0 Value
 *                             Err_recovery_action
 *                             0 = Pass name to Exit Pgm with ErrID
 *                             1 = Bypass the object
 *                             2 = Send CPDA1C0 to Joblog
 *                             3 = Pass NULL to Exit Pgm with ErrID
 *                             4 = End Process
D  UserFunction@                 *   Value
 *                             UserFunction_ptr, see DS QP0LUF
D  FuncCtrlBlk@                  *   Value
 *                             Function_ctlBlk_ptr, see FuncCtrlBlk
```

Here is an abstract from our sample program where we define the variables used by the API, the return value field, and the call statement:

```
 * Qp0lProcessSubtree prototypes                                   2
 /COPY RPGISCOOL/SWEEPSRC,PRCSUBTRPR
...
 * Path name structure based from QSYSINC/QRPGLESRC,QLG            3
D QLGPN           DS
 /COPY RPGISCOOL/SWEEPSRC,PATHNAMEDF
D  QLGPN00                     8000A

 * Array size constants for structure definitions                  4
D QP0TYPES_C      C                   66
D Qlg_Path_NameC  C                   10
 * Qp0lProcessSubtree Type Definitions
 /COPY RPGISCOOL/SWEEPSRC,PRCSUBTRDF
 * For Array of pointers initialization
D InExclusionPN   S                   Like(QLGPN) DIM(Qlg_Path_NameC)

 * Field definition for parameters of Qp0lProcessSubtree        5
D PathName@       S               *   Inz(%addr(PathName))
D  PathName       S                   LIKE(QLGPN)
D SubtreeLevel    S             10U 0 INZ(0)
D ObjtypeArray@   S               *   Inz(%addr(QP0LOL))
D LocalRemote     S             10U 0 INZ(0)
D InExclusion@    S               *   Inz(%addr(QP0INEXL))
D ErrRecovery     S             10U 0 INZ(0)
D UserFunction@   S               *   Inz(%addr(QP0LUF))
D FuncCtrlBlk@    S               *   Inz(%addr(FuncCtrlBlk))
...
 * Stand-Alone variables                                       6
D Rc              S             10I 0 INZ(0)
...
 * Calling the Qp0lProcessSubtree API, receiving return code in RC 16
C                   Eval      Rc = ProcessSubtree(PathName@ :
C                                    SubtreeLevel : ObjtypeArray@ :
C                                    LocalRemote : InExclusion@ :
C                                    ErrRecovery : UserFunction@ :
C                                    FuncCtrlBlk@)
```

We initialized the non-pointer type field to their default on the field declaration, so we do not have to refer to them in the C-specs. The object types array, the In_exclusion list, and the function control block parameters can be initialized to *NULL if they do not specify any values.

The usage of the different parameters involving pointers are discussed in the following sections.

Path name

The API uses the path name structure to pass the path name as a parameter to the API. Only a pointer to the path name structure defined in your program is passed to the API.

Refer to 5.7.3.1, “Path name structure or path name format” on page 254, for a description of the path name structure.

The path name is formatted and passed as a pointer to the API in the sample program using these instructions:

```
 * Input parameters for program                                1
D Sweep           PI
D  InputPath                   100A
...
 * Path name structure based from QSYSINC/QRPGLESRC,QLG        3
D QLGPN           DS
 /COPY RPGISCOOL/SWEEPSRC,PATHNAMEDF
D  QLGPN00                    100A
...
 * Field definition for parameters of Qp0lProcessSubtree        5
D PathName@       S               *   Inz(%addr(PathName))
D  PathName       S                   LIKE(QLGPN)
..
 * Initialize the Path Name constants                           7
C                   Eval      QLGCCSID02 = 37
C                   Eval      QLGCID = 'US'
C                   Eval      QLGLID = 'ENU'
C                   Eval      QLGERVED07 = *ALLx'00'
C                   Eval      QLGPT = 0
C                   Eval      QLGPND = x'6100'
C                   Eval      QLGRSV200 = *ALLx'00'
 * Initialize the Input Path Name variable                      8
C                   If        InputPath <> *blank
C                   Eval      QLGPL = %len(%trim(InputPath))
C                   Eval      QLGPN00 = %subst(InputPath : 1 : QLGPL)
C                   Eval      PathName = QLGPN
 * No path name specified
```

```
C                   Else
C                   Eval      PathName@ = *NULL
C                   Endif
```

In this example, we used the default country, language, and code character set IDs of US, ENU and 00037. Of course, this can be modified to reflect your job attributes or retrieved dynamically by the program. We kept the initialization of these constants in the C-specs so the /COPY member can be used for a different environment. We also chose to limit the path name size to 100 bytes as the input parameter definition.

The *InputPath* field contains the starting path name, which must be a directory. This parameter is mandatory unless an Inclusion list is passed (see “IN_Exclusion list” on page 286 for more information). The path name is copied into the *QLGPN00* path name variable of the path name structure, and the length of the path name is placed into *QLGPL*. Then, the complete structure definition (QLGPN) is copied into the *Path Name* variable so the memory location of this variable can be passed to the API as a pointer. The parameter is initialized to **NULL* if no path name is passed to the program.

Array of object types

This parameters identifies a list of object types that will make the API invoke the process for a path name exit program if they are encountered during the process. The default, a NULL pointer, as used in our sample program, indicates that *ALL object types will be processed.

The following structure is provided to pass the object types list to the API. A pointer to the structure definition (DS) is passed to the API. This structure definition is based on the one found in the member QP0LSTDI of the source file QSYSINC/QRPGLESRC from the System openness include licensed program:

```
 * PRCSUBTRDF from SWEEPSRC in RPGISCOOL (part 1 of 3)
 * Qp0lProcessSubtree() API Object types list structure def.    4
DQP0LOL           DS
D QP0NBROO00                    10I 0 INZ(0)
 *                                  Number of object
 *                                    types in the list
D QP0TYPES                      11    DIM(QP0TYPES_C)
 *                                 Variable length entry
```

As you can see, the dimension or number of elements of the table are defined using a constant. This design allows the structure definition to be fixed and placed in an external member and set into the main source by using the /COPY method when needed. The size constant is defined and initialized in the main source depending on the intended utilization of this array.

The object type array is formatted and passed as a pointer to the API in the sample program using these instructions:

```
 * Input parameters for program                                 1
D Sweep           PI
...
D  ObjTypes                    662A
...
 * Array size constants for structure definitions               4
D QP0TYPES_C      C                   66
 * Qp0lProcessSubtree Type Definitions
 /COPY RPGISCOOL/SWEEPSRC,PRCSUBTRDF
...
 * Field definition for parameters of Qp0lProcessSubtree        5
D ObjtypeArray@   S               *   Inz(%addr(QP0LOL))
...
 *Stand-Alone fields                                            7
D                 DS
```

```
D ObjTypes2               1      2A
D ObjTypesN                      5I 0 Overlay(ObjTypes2)
D Idx             S              5P 0
...
 * Initialize Object Type Array to *NULL if no ObjType          10
C                   Eval      ObjTypes2 = %subst(ObjTypes : 1 : 2)
C                   Eval      QP0NBROO00 = ObjTypesN

C                   For       Idx = 1 to QP0NBROO00
C                   Eval      QP0TYPES(Idx) = %subst(ObjTypes :
C                                         (10 * idx) -7 : 10)
 * Add Null character if no special values used
C                   If        QP0TYPES(Idx) <> '*ALLSTMF' and
C                             QP0TYPES(Idx) <> '*ALLDIR' and
C                             QP0TYPES(Idx) <> '*MBR'
C                   Eval      QP0TYPES(Idx) = QP0TYPES(Idx) + x'00'
C                   EndIf
C                   EndFor
 * Map pointer to *NULL for default
C                   If        QP0NBROO00 =  0 or QP0TYPES(1) = '*ALL'
C                   Eval      ObjtypeArray@ = *NULL
C                   End
```

The ObjTypes input parameters are defined as a list of 66 elements of 10 bytes each. There 66 possible object type choices at V4R4 on the AS/400 system. This list of parameters also include two bytes (at the beginning) to indicate the number of elements passed to the program. This number is extracted in ObjTypesN and then placed in QP0NBROO00, as defined in the QP0LOL structure definition.

The Object type array in the structure definition, QP0TYPES, is defined using the constant QP0TYPES_C for the size of the array of 11 bytes for each element. This last arrays contains one more byte per elements since the null termination string is required except when using the three special values (*ALLDIR, *ALLSTMF, and *MBR). This array is filled by extracting the value from the ObjTypes input parameter.

Because the API only requires a pointer to the structure definition, the field ObjtypeArray@ is used to pass the memory location as a pointer. If no object types have been passed to the program or *ALL object types was specified, the API requires a *NULL value for the pointer parameter.

IN_Exclusion list

This parameters represents a list of the only directories (path name) that need to be included in the API sweep, or a list of directories that needs to be excluded from the sweep when starting at the input path name. If an Inclusion list is specified, the input path name must be blank. This parameter implies that the usage of an array of pointer techniques as the path name specified are passed as pointer referring to a path name structure definition. Multiple directories can be passed to the API as an Inclusion or an Exclusion list.

The following structure definition is provided to pass the Inclusion or Exclusion list to the API. A pointer to this structure definition is passed to the API. This structure definition is based on the one found in the member QP0LSTDI of the source file QSYSINC/QRPGLESRC from the System openness include licensed program:

```
 * PRCSUBTRDF from SWEEPSRC in RPGISCOOL (part 2 of 3)
 * Qp0lProcessSubtree() API IN_EXclusion list structure def.       4
DQP0INEXL         DS
D QP0INEX                       10I 0 INZ(0)
 *                                  Inclusion list or
 *                                    exclusion list
 *                                    type identifier
D QP0NBROP                      10I 0 INZ(0)
```

```
 *                                  Number of path name
 *                                    pointers in the
 *                                    inclusion or
 *                                    exclusion list
D QP0ERVED18                     8    INZ(*ALLx'00')
 *                                  Must be zero
 *
D Qlg_Path_Name                   *   DIM(Qlg_path_NameC)
 *                                 Variable length entry
 *                                 Array of pointers
```

The dimension or size of the Qlg_Path_Name array is defined using a constant, which should be specified in the main source, where the /COPY member instruction using this structure definition is specified. Same as the Object Types array, this design allows flexibility as the /COPY member can stay constant even if the number of elements of the array can be flexible in different programs.

The In_exclusion list parameter is formatted and passed to the API in the sample program using those instructions:

```
 * Input parameters for program                                 1
D Sweep           PI
...
D  In_ExclusionT                 1A
D  In_ExclusionP              1002A
...
 * Path name structure based from QSYSINC/QRPGLESRC,QLG         3
D QLGPN           DS
 /COPY RPGISCOOL/SWEEPSRC,PATHNAMEDF
D  QLGPN00                     100A

 * Array size constants for structure definitions               4
D Qlg_Path_NameC  C                   10
 * Qp0lProcessSubtree Type Definitions
 /COPY RPGISCOOL/SWEEPSRC,PRCSUBTRDF
 * For Array of pointers initialization
D InExclusionPN   S                   Like(QLGPN) DIM(Qlg_Path_NameC)
...
 * Field definition for parameters of Qp0lProcessSubtree        5
D InExclusion@    S               *   Inz(%addr(QP0INEXL))
...
 * Stand-Alone fields                                           7
D                 DS
D InExcl2                1      2A
D InExclN                      5I 0 Overlay(InExcl2)
D Idx             S            5P 0
...
 * Initialize the Path Name constants                           7
C                   Eval      QLGCCSID02 = 37
C                   Eval      QLGCID = 'US'
C                   Eval      QLGLID = 'ENU'
C                   Eval      QLGERVED07 = *ALLx'00'
C                   Eval      QLGPT = 0
C                   Eval      QLGPND = x'6100'
C                   Eval      QLGRSV200 = *ALLx'00'
...
 * Initialize In_Exclusion parameter                            12
C                   If        In_ExclusionT <> *blank
C                   Move      In_ExclusionT QP0INEX
C                   Eval      InExcl2 = %subst(In_ExclusionP : 1 : 2)
C                   Eval      QP0NBROP = InExclN

C                   For       Idx = 1 to QP0NBROP
C                   Eval      QLGPN00 = %subst(In_ExclusionP :
C                                    (100 * idx) - 97 : 100)
C                   Eval      QLGPL = %len(%trim(QLGPN00))
C                   Eval      InExclusionPN(Idx) = QLGPN
C                   Eval      Qlg_Path_Name(Idx) =
C                                %addr(InExclusionPN(Idx))
C                   EndFor
C                   EndIf
C                   If        QP0NBROP = 0
C                   Eval      InExclusion@ = *NULL
C                   Endif
```

Since the API receives two types of lists (Inclusion or Exclusion) and they are mutually exclusive, two input parameters are defined to be passed to the program: the list type and the path names list. The list type parameter In_ExclusionT is passed to QP0INEX unless it contains a blank meaning that no lists have been passed to the program. Same as for the object type arrays, the path names list is passed as a maximum of 10 values, each having a length of 100 bytes (the number of values and the length are limitations of this sample program only). This list is passed in the parameter In_ExclusionP, with a length of 1002 bytes since the first two bytes indicate the number of values passed, which is extracted to QP0NBROP using InExclN and InExcl2 as intermediate.

Using the same techniques as for the Input Path Name parameter, the path names are extracted from In_ExclusionP and then incorporated in the path name structure with the proper length and other constants initialized. As an array of pointers is used to pass the path names to the API, each path name encountered is copied into a path name array, InExclusionPN, using the path name structure as a definition. The memory location of each filled index in the path name array is passed as a pointer to the array of pointers Qlg_Path_Name within the In_Exclusion list structure definition (QP0INEXL).

Since the API requires a pointer to the structure definition, the field InExclusion@ is used to pass the memory location of the In_Exclusion list structure definition as a pointer. If the number of elements of the path names list is zero, the pointer is initialized to a *NULL value as required by the API.

User function pointer

The user function pointer represents a pointer to a structure definition where a program name and library or a procedure pointer are specified. This program or procedure is referred as the *process a path name* exit program, which is called by the API every time an object meeting the selection criteria is encountered. The exit program (or exit procedure) is explained in 5.7.5.3, “Process a path name exit program” on page 290.

The following structure definition is provided to pass the exit program name or procedure pointer to the API. A pointer to the structure definition is required by the API. This structure definition is based on the one found in the member QP0LSTDI of the source file QSYSINC/QRPGLESRC from the System openness include licensed program:

```
 * PRCSUBTRDF from SWEEPSRC in RPGISCOOL (part 3 of 3)
 * Qp0lProcessSubtree() API user function structure def.         4
DQP0LUF          DS
 *                              Qp0l User Function type
D QP0FT                        10I 0 INZ(0)
 *                                 Procedure or program
 *                                   type flag
D QP0BRARY                     10    INZ(*BLANK)
 *                                 Program library
D QP0OGRAM                     10    INZ(*BLANK)
 *                                 Program name
D QP0HDACN                      1    INZ(X'00')
 *                                 Multithread action
D QP0ERVED19                    7    INZ(*ALLx'00')
 *                                 must be zero
D QP0LPS                         *   PROCPTR
 *                                 Procedure pointer
```

The Function type flag *QP0FT* must be initialized to zero if a subprocedure is used as the exit program. Or, QP0FT must be initialized if a program is used instead. The program name/library and the procedure pointer fields are mutually exclusive. The User function parameter is formatted and passed to the API in the sample program using the following instructions:

```
 * Qp0lProcessSubtree prototypes                                   2
 /COPY RPGISCOOL/SWEEPSRC,PRCSUBTRPR
...
 * Qp0lProcessSubtree Type Definitions                             4
 /COPY RPGISCOOL/SWEEPSRC,PRCSUBTRDFprcsubtrdf
...
 * Field definition for parameters of Qp0lProcessSubtree           5
D UserFunction@   S              *   Inz(%addr(QP0LUF))
...
 * Initialize the Procedure Pointer with the pointer               14
C                   Eval      QP0FT = 0
C                   Eval      QP0LPS = %paddr('SweepExit')
 * Initialize the UserFunction to UserFunction structure.
C                   Eval      UserFunction@ = %addr(QP0LUF)
...
```

Because our sample program is using a subprocedure for the process a path name exit program, the prototype of the subprocedure must be defined in the main source. In our example, the member PRCSUBTRPR contains the subprocedure prototype listed in 5.7.5.3, “Process a path name exit program” on page 290.

You can find more information on the process a path name exit program (or exit subprocedure) and its prototype in the following section. In this example, the subprocedure used as an exit program of the API is exported and is part of a different module. This module needs to be bound to the main module at the program creation. The subprocedure can also be part of the main module or placed in a service program. Only the prototype allows the memory location of the subprocedure to be mapped to the pointer parameter required by the structure definition.

Function control block

The function control block is a memory location where any kind of data can be passed from the API calling program to the exit program. This parameter is mainly used for communication between the calling program and the exit program and not processed by the API. For this sample program, we passed a 100-byte data variable directly from the input parameter to the exit program.

The data is passed to the API using the memory location of the variables as a pointer. The Function Control block parameter is formatted and passed to the API in the sample program using the following instructions:

```
 * Input parameters for program                                      1
D Sweep           PI
...
D  FuncCtrlBlk                 100A
...
 * Field definition for parameters of Qp0lProcessSubtree              5
D FuncCtrlBlk@    S              *   Inz
...
 * Initialize the Function control block pointer                     15
C                   Eval      FuncCtrlBlk@ = %addr(FuncCtrlBlk)
```

As the API requires a pointer, the memory location of the input parameters FuncCtrlBlk is passed as a pointer in variable FuncCtrlBlk@. This parameter can be initialized to a null value if no values are required to be passed between the API and the process of a path name exit program.

Please refer to the Qp0lProcessSubtree() API documentation in *System API Reference OS/400 UNIX-Type APIs,* SC41-5875, in the chapter "Integrated File

System APIs", for more information on thoses parameters and the regular ones not using pointers.

5.7.5.3 Process a path name exit program

The Process a path name exit program is a user-specified exit program that is called by the Qp0lProcessSubtree() function for each object in the API's search that meets the caller's selection criteria. This exit program can be a subprocedure or a program. In our sample program, we used a subprocedure placed in an external module as the API exit program. This subprocedure will be invoked for each object selected by the API and prints the path name along with the function control block text passed by the API in a spooled file. The subprocedure also handles a spooled file header and footer, plus errors which occur during the object selection.

Table 75 shows the parameters for this function. These parameters must be defined in the prototype used to call this API.

Table 75. Required parameters for the Process a path name exit program

Argument	Description	Use	RPG data type	C data type
SltStatus	Selection Status Pointer	Input	*	Binary(4)
ErrorValue	Error Value pointer	Input	*	Binary(4)
ReturnValue	Return Value pointer	Output	*	Binary(4)
PathName	Object name pointer	Input	*	Char(*)
FuncCtrlBlk	Function control block pointer	Input	*	Char(*)

Here’s an example of the prototype used to map thoses parameters:

```
 * PRCSUBTRPR from SWEEPSRC in RPGISCOOL (part 2 of 2)
 * Process a Path name Exit Pgm prototype                          2
 * Defined as per System API Reference - OS/400 UNIX-Type APIs manual
D SweepExit       PR                  ExtProc('SweepExit')
D  SelectStatus                   *   Value
D  ErrorValue                     *   Value
D  ReturnValue                    *   Value
D  ObjectName                     *   Value
D  FCBPointer                     *   Value
```

All of these parameters are pointers passed by the API. In our sample program, we map the memory locations using the following instructions:

```
 * Qp0lProcessSubtree prototypes                                 19
 /COPY RPGISCOOL/SWEEPSRC,PRCSUBTRPR
...
 * Subprocedure declaration                                      19
D SweepExit       PI
D  SltStatus@                     *   Value
D  ErrorValue@                    *   Value
D  ReturnValue@                   *   Value
D  PathName@                      *   Value
D  FuncCtrlBlk@                   *   Value

 * Path name structure based from QSYSINC/QRPGLESRC,QLG        20
D QLGPN           DS                  Based(PathName@)
 /COPY RPGISCOOL/SWEEPSRC,PATHNAMEDF
D  QLGPN00                        *
D Object          S          16000C   Based(QLGPN00)

 * Input parameters                                             21
D SltStatus       S             10I 0 BASED(SltStatus@)
D ErrorValue      S             10I 0 BASED(ErrorValue@)
D FuncCtrlBlk     S            100A   BASED(FuncCtrlBlk@)
```

```
 * Output parameter
D ReturnValue     S             10I 0 INZ(0)
...
 * Return value pointer initialization                      26
C                   Eval      ReturnValue@ = %addr(ReturnValue)
```

The parameters used are explained in the following sections.

Selection status and error value

The selection status parameter contains the value of zero when the exit program is invoked without any problems encountered during the selection done by the API. After the last object was processed, the API calls back the exit program with a value of 1 in the selection status parameter indicating that it is the last call to the exit program. The selection status parameter has other values in case of problems encountered during the selection processing. The *errorvalue* pointer is used to indicate the error number.

Path name

The path name pointer passed to the exit program contains the path name of the object selected by the API to be processed. This memory location needs to be mapped to the path name structure, as defined in 5.7.3.1, “Path name structure or path name format” on page 254. Three particularities to the path name passed to the exit program are:

- The path name within the path name structure contains a pointer to a memory location where the path name text is specified. This memory location needs to be mapped to a path name variable, as *Object*, in our example, with a sufficient variable length to contain the path name.
- The path name passed by the API is always in a Unicode character set (UCS-2) since IFS APIs handle path names in Unicode. The environment attributes of the path name structure (CCSID, country ID, and language ID) reflect the path name structure. As Unicode is a double-byte character set, the variable length needs to be double compared to a regular one-byte character field.
- The path name length also reflects the size of the path name in unicode, as twice as high as if it would be a regular character field. The %subst function (all the string functions, in fact) in RPG IV expects the length to be the number of characters, not the number of bytes. If you have a Unicode length measured in bytes, it must be divided by 2 to get the number of Unicode characters, as shown in the following example:

```
C                   Eval      Object132 = %subst(Object : 1 :
C                                          %div(QLGPL:2))
```

Unicode support in RPG IV

Here is an abstract of the *ILE RPG for AS/400 Reference*, SC09-2508, which lists the V4R4 enhancements regarding the support of the Universal Character Set Version 2 (UCS-2) or Unicode support in RPG IV:

- The UCS-2 (Unicode) character set can encode the characters for many written languages. The field is a character field whose characters are two bytes long.
- By adding support for Unicode, a single application can now be developed for a multinational corporation, minimizing the necessity to perform code page conversion. The use of Unicode permits the processing of characters in multiple scripts without the loss of integrity.
- Support for conversions between UCS-2 fields and graphic or single-byte character fields using the MOVE and MOVEL operations, and the new %UCS2 and %GRAPH built-in functions.
- Support for conversions between UCS-2 fields or graphic fields with different Coded Character Set Identifiers (CCSIDs) using the EVAL, MOVE, and MOVEL operations, and the new %UCS2 built-in function.

As discussed, RPG IV handles a Unicode character string at V4R4. If you are running an earlier version of OS/400 and receiving Unicode data from the IFS APIs, you need to convert the data using a national language conversion API. The API is iconv()–Code Conversion API, which needs to be used in conjunction with QtqIconvOpen()–Code Conversion Allocation API and iconv_close()–Code Conversion Deallocation API. These APIs are documented in *System API Reference OS/400 National Language Support APIs*, SC41-5863.

Return value
The return value pointer needs to map to a memory location represented by a 4 byte binary or a 10-digit signed integer value variable. The value of this variable can be 0 for a successful completion of the exit program, -1 for successful completion, but the API should skip the rest of the objects in the current directory, or greater than zero if an error occurred in the exit program.

Function control block
The function control block pointer maps to a memory location where the data has been stored by the program calling the API. The pointer parameter should be mapped to the same variable type and length as the original one, although data type compatibility is allowed.

5.7.5.4 Anatomy of the sample program SWEEP
As described in 5.7.5.1, “Overall concept of the sample program SWEEP” on page 280, this sample program scans directories on the Integrated File System and calls an exit program for any object found that matches the selection criteria. The selection criteria is specified using a command object. The output of the program is a spooled file listing all the objects found.

Main program outline

Here is the outline of the Sweep main program used for our sample program:

- **Field declarations**

  1 Declare the prototype and procedure interface for the input parameters.

  2 Declare the Qp0lProcessSubtree API prototype for the API interface

  3 Include the path name structure as a data structure where the header and path name field are defined in the main procedure. For more information, refer to 5.7.3.1, "Path name structure or path name format" on page 254.

  4 Include the structure definitions for some of the pointer type parameters of the Qp0lProcessSubtree API such as the object type array, the In_Exclusion list, and the User function parameters. The size of the arrays included within these structure definitions are defined by using constant fields declared in the main procedure.

  5 Declare a variable used to store the API parameters.

  6 Declare the prototype used to call the DspError subprocedure in case it occurred during API processing.

  7 Declare standalone fields for numeric to character translation, among other types, and initialize the path name constant.

- **Main process**

  8 Initialize the input path name parameter and the associated length into a *Pathname* variable so the memory location can be passed to the API using a pointer. For more information, please refer to "Path name" on page 284.

  9 Initialize the *Subtreelevel* unsigned numeric parameter with the *Subtreeparm* signed numeric parameter passed by the command.

  10 Initialize the object type array pointer with the object type information passed by the command, using the object type structure definition. If no specific object types were requested, the pointer is initialized to a *null* value. For more information, refer to "Array of object types" on page 285.

  11 Initialize the *LocalRemote* numeric parameter by using the *LocalRmt* character variable passed by the command.

  12 Initialize the In_Exclusion list pointer with the Inclusion or Exclusion list passed by the command, using the In_Exclusion list structure definition. This parameters uses an array of pointers mapped to the array *InExclusionPN* in the main procedure. If no list of objects is requested, the pointer is initialized to a *null* value. For more information, refer to "IN_Exclusion list" on page 286.

  13 Initialize the *Errrecovery* numeric parameter using the *ErrorRcv* character variable passed by the command.

  14 Initialize the User function pointer with the %paddr built-in function on the subprocedure name as declared in the subprocedure prototype. The exit program type parameter *Qp0FT* is also initialized and the memory location of the structure definition *Qp0LUF* is passed to the parameter as a pointer. For more information, refer to "User function pointer" on page 288.

  15 The Function control block pointer parameter is initialized with the memory location of the user text passed by the command. For more information, refer to "Function control block" on page 289.

16 The Qp0lProcessSubtree is called and receives the return value in the variable *Rc*.

17 Display the proper message depending on the return value content. For more information on the return value, check the exit program at 26.

Exit program outline

Here is the outline of the *SweepExit* subprocedure used as the exit program is used in our sample program:

- **File and fields declarations**

  18 Printer file declaration.

  19 Subprocedure prototype and procedure interface for input parameters of the exit program.

  20 Include the path name structure as a data structure, where the header and path name field are defined in the main procedure. The path name is defined as a pointer and the variable *Object* of type Unicode is used to map the memory location. For more information, refer to 5.7.3.1, "Path name structure or path name format" on page 254.

  21 Declare a variable used to store the pointer parameters received from the API and the return value pointer passed back to the API. The return value is initialized to 1 (error occurred) and changes to zero upon successful completion of the exit program.

  22 Initialize standalone fields used in the subprocedure, mainly in the spooled file produced by the exit program. The *HeaderPrint* variable has been declared as *Static*, so it will exist and keep its value if the exit program is called again (instead of being initialized every time the exit program is called).

- **Main process**

  23 Print the header of the spooled file on the first execution of the subprocedure.

  24 If no error is reported by the API, retrieve the first 132 bytes of the path name from the path name structure into unicode format. Move the 132 bytes path name and the function control block text into print variables and print the detail entry. Refer to 5.7.5.3, "Process a path name exit program" on page 290, for more information on Unicode support.

  25 If it is the last call to the exit program, it initializes the successful message text to print in the footer. In case an error occurred in the object selection, retrieve the error message using the *strerror* C runtime function, display the error message, and print the error text. In both cases, print the footer, and close the opened file.

  26 Initialize and return the successful return value to the API. In a more complex program code, the successful return value is initialized after an error validation function.

The sample code

The following section includes the complete sample code used in the Sweep sample program. This sample application is made of multiple modules, a printer file, and multiple /COPY members, which are described as follows:

- The main module
- The exit program subprocedure module
- The Exit program printer file
- The User created command object
- The main module prototype (/COPY member)
- The path name structure definition (/COPY member)
- The prototypes for Qp0lProcessSubtree API (/COPY member)
- The Qp0lProcessSubtree API structure definitions
- The DspError subprocedure prototype
- The DspError subprocedure module

Main SWEEP module

The filename for the main module is SWEEP. Refer to the previous sections for explanations of the code markers:

An exercise for you

Notice that this program uses the commercial at (@) to indicate that the variable is a pointer. This breaks the style guide rule found in 2.1.3.3, "Avoid using special characters (for example, @, #, $) when naming items" on page 22.

Avoid 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, for example, pCustNbr or cActStsCde.

```
 * SWEEP from IFSSRC in RPGISCOOL
*******************************************************************
 * Module name: SWEEP
 *
 * This program demonstrate the usage of the Qp0lProcessSubtree API
 *  which scan the Integrated File system on selection criteria
 *  passed to the program by a command object. The API will call
 *  a subprocedure as the API Exit Program (Process a path Name Exit
 *  program), which will print the object found.
 *
 * *** Module needs to be bind with modules DSPERROR and SWEEPEXIT
 *     on program creation
*******************************************************************

H option(*SRCSTMT) bdndir('QC2LE')

 * Program prototypes                                              1
 /COPY RPGISCOOL/SWEEPSRC,SWEEPPR

 * Input parameters for program                                    1
D Sweep           PI
D  InputPath                   100A
D  SubtreeParm                   1S 0
D  ObjTypes                    662A
D  LocalRMT                      1A
D  In_ExclusionT                 1A
D  In_ExclusionP              1002A
D  ErrorRcv                      1A
D  FuncCtrlBlk                 100A

 * Path name structure based from QSYSINC/QRPGLESRC,QLG              3
D QLGPN           DS
 /COPY RPGISCOOL/SWEEPSRC,PATHNAMEDF
D  QLGPN00                     100A

 * Qp0lProcessSubtree prototypes                                     2
 /COPY RPGISCOOL/SWEEPSRC,PRCSUBTRPR

 * Array size constants for structure definitions
```

```
D QP0TYPES_C      C                   66
D Qlg_Path_NameC  C                   10
 * Qp0lProcessSubtree Type Definitions                                  4
 /COPY RPGISCOOL/SWEEPSRC,PRCSUBTRDF
 * For Array of pointers initialization
D InExclusionPN   S                   Like(QLGPN) DIM(Qlg_Path_NameC)

 * Field definition for parameters of Qp0lProcessSubtree                5
D PathName@       S               *   Inz(%addr(PathName))
D  PathName       S                   LIKE(QLGPN)
D SubtreeLevel    S             10U 0 INZ(0)
D ObjtypeArray@   S               *   Inz(%addr(QP0LOL))
D LocalRemote     S             10U 0 INZ(0)
D InExclusion@    S               *   Inz(%addr(QP0INEXL))
D ErrRecovery     S             10U 0 INZ(0)
D UserFunction@   S               *   Inz(%addr(QP0LUF))
D FuncCtrlBlk@    S               *   Inz

 * Prototypes requires for DspError subprocedure                        6
 /COPY RPGISCOOL/SWEEPSRC,ERRPROTO

 *Stand-Alone fields                                                    7
D                 DS
D ObjTypes2               1      2A
D ObjTypesN                      5I 0 Overlay(ObjTypes2)
D                 DS
D InExcl2                 1      2A
D InExclN                        5I 0 Overlay(InExcl2)

D SuccessMsg      C                   'Qp0lProcessSubtree() Successful'
D UnsuccessMsg    C                   'ERROR on Qp0lProcessSubtree()'

D Rc              S             10I 0 INZ(0)
D Idx             S              5P 0

 * Initialize the Path Name constants
C                   Eval      QLGCCSID02 = 37
C                   Eval      QLGCID = 'US'
C                   Eval      QLGLID = 'ENU'
C                   Eval      QLGERVED07 = *ALLx'00'
C                   Eval      QLGPT = 0
C                   Eval      QLGPND = x'6100'
C                   Eval      QLGRSV200 = *ALLx'00'

 * Initialize the Input Path Name variable                               8
C                   If        InputPath <> *blank
C                   Eval      QLGPL = %len(%trim(InputPath))
C                   Eval      QLGPN00 = %subst(InputPath : 1 : QLGPL)
C                   Eval      PathName = QLGPN
 * No path name specified
C                   Else
C                   Eval      PathName@ = *NULL
C                   Endif

 * Subtree parameter initialization                                      9
C                   Eval      SubtreeLevel = SubtreeParm

 * Initialize Object Type Array to *NULL if no ObjType                  10
C                   Eval      ObjTypes2 = %subst(ObjTypes : 1 : 2)
C                   Eval      QP0NBROO00 = ObjTypesN

C                   For       Idx = 1 to QP0NBROO00
C                   Eval      QP0TYPES(Idx) = %subst(ObjTypes :
C                                        (10 * idx) -7 : 10)
 * Add Null character if no special values used
C                   If        QP0TYPES(Idx) <> '*ALLSTMF' and
C                             QP0TYPES(Idx) <> '*ALLDIR' and
C                             QP0TYPES(Idx) <> '*MBR'
C                   Eval      QP0TYPES(Idx) = QP0TYPES(Idx) + x'00'
C                   EndIf
C                   EndFor
 * Map pointer to *NULL for default
C                   If        QP0NBROO00 =  0 or QP0TYPES(1) = '*ALL'
C                   Eval      ObjtypeArray@ = *NULL
C                   End

 * LocalRemote parameter initialization                                 11
C                   Move      LocalRmt      LocalRemote
```

```
 * Initialize In_Exclusion parameter                                    12
C                   If        In_ExclusionT <> *blank
C                   Move      In_ExclusionT QP0INEX
C                   Eval      InExcl2 = %subst(In_ExclusionP : 1 : 2)
C                   Eval      QP0NBROP = InExclN
C                   For       Idx = 1 to QP0NBROP
C                   Eval      QLGPN00 = %subst(In_ExclusionP :
C                                         (100 * idx) - 97 : 100)
C                   Eval      QLGPL = %len(%trim(QLGPN00))
C                   Eval      InExclusionPN(Idx) = QLGPN
C                   Eval      Qlg_Path_Name(Idx) =
C                                   %addr(InExclusionPN(Idx))
C                   EndFor
C                   EndIf
C                   If        QP0NBROP = 0
C                   Eval      InExclusion@ = *NULL
C                   Endif

 * Error Recovery parameter initialization                              13
C                   Move      ErrorRcv      ErrRecovery

 * Initialize the Procedure Pointer with the pointer of the subprocedure
C                   Eval      QP0FT = 0
C                   Eval      QP0LPS = %paddr('SweepExit')
 * Initialize the UserFunction to UserFunction structure.                14
C                   Eval      UserFunction@ = %addr(QP0LUF)

 * Initialize the Function control block pointer                         15
C                   Eval      FuncCtrlBlk@ = %addr(FuncCtrlBlk)

 * Calling the Qp0lProcessSubtree API, receiving return code in RC       16
C                   Eval      Rc = ProcessSubtree(PathName@ :
C                                       SubtreeLevel : ObjtypeArray@ :
C                                       LocalRemote : InExclusion@ :
C                                       ErrRecovery : UserFunction@ :
C                                       FuncCtrlBlk@)

C                   If        RC = 0
 * Successful execution                                                   17
C     Successmsg    Dsply
C                   Else
 * Display error msg
C                   CallP     DspError('PRCSubTree')
C     UnsuccessMsg  Dsply
C                   EndIf

 * End of program
C                   Eval      *InLr = *On
```

The SWEEPEXIT exit program

The filename for the exit program module is SWEEPEXIT. Refer to the previous sections for explanations of the code markers:

```
 * SWEEPEXIT from IFSSRC in RPGISCOOL
 * Subprocedure for exit program of the Qp0lProcessSubtree API

H NoMain option(*SRCSTMT) bnddir('qc2le')

 * Printer file external definition                                     18
FSweepPrtf O    E             PRINTER OFLIND(*In50)

 * Qp0lProcessSubtree prototypes
 /COPY RPGISCOOL/SWEEPSRC,PRCSUBTRPR

 * Subprocedure declaration                                             19
P SweepExit       B                   Export

D SweepExit       PI
D  SltStatus@                     *   Value
D  ErrorValue@                    *   Value
D  ReturnValue@                   *   Value
D  PathName@                      *   Value
D  FuncCtrlBlk@                   *   Value

 * Path name structure based from QSYSINC/QRPGLESRC,QLG                  20
D QLGPN           DS                  Based(PathName@)
```

```
 /COPY RPGISCOOL/SWEEPSRC,PATHNAMEDF
D  QLGPN00                        *
D Object          S          16000C   Based(QLGPN00)

 * Input parameters                                                     21
D SltStatus       S             10I 0 BASED(SltStatus@)
D ErrorValue      S             10I 0 BASED(ErrorValue@)
D FuncCtrlBlk     S            100A   BASED(FuncCtrlBlk@)

 * Output parameter
D ReturnValue     S             10I 0 INZ(1)

 * Stand-ALone fields                                                   22
D Object132       S            264C   Inz(*BLANK)
D SuccessMsg      C                   'End of report'
D UnsuccessMsg    C                   'An error has occured...'
D HeaderPrint     S               N   Static Inz(*off)

D errorMsg@       S               *   Inz
D  errorMsg       S            100A   Based(errorMsg@)
D errortxt        S             52A   Inz(*blank)

 * Prototypes requires for strError function
 /Copy RPGISCOOL/IFSSRC,ERRPROTO

 * If first time called or page overflow, print header                  23
C                   If        not HeaderPrint or *in50 = *on
C                   Write     SWEEPH
C                   Eval      HeaderPrint = *On
C                   Eval      *in50 = *off
C                   EndIf

 * Selection Status = 0 (QP0L_SELECT_OK)
C                   If        SltStatus = 0

 * If ObjectName@ contains a valid pointer, substring                   24
 *  for non-unicode length
C                   If        PathName@ <> *NULL
C                   Eval      Object132 = %subst(Object : 1 :
C                                           %div(QLGPL:2))

 * Conversion from UCS-2 (Unicode) to single byte character field
 *  done by MOVE at V4R4
 * (Shown as example only as RPG IV is able to print an UCS-2 type field)
C                   Movel     Object132     ObjectPrt
C                   Movel     FuncCtrlBlk   Text
C                   Else

 * If ObjectName@ doesn't contains pointer
C                   Eval      ObjectPrt = 'Null pathname'
C                   EndIf

 * Print Function Control Block passed from main and pathname
C                   Write(E)  SWEEPD

 * Select Status isn't OK
C                   Else

 * Selection is done (last call)                                        25
C                   If        SltStatus = 1
C                   Eval      Text = SuccessMsg
C                   Else

 * An error has occured during the selection,
 *  retrieve and display the error message
C                   Eval      errorMsg@ = StrError(ErrorValue)
C                   Eval      errortxt = 'SweepExit' + '->' +
C                                         %char(ErrorValue)
C                             + ':' + %subst(errormsg : 1 : 37)
C     errortxt      Dsply
C                   Eval      Text = UnSuccessMsg
C                   EndIf

 * Writing Footer and closing printer file
C                   Write(E)  SweepE
C                   Close(E)  SweepPrtf

C                   Endif
```

```
 * Return value pointer initialization                                           26
C                   Eval      ReturnValue = 0
C                   Eval      ReturnValue@ = %addr(ReturnValue)

P                 E
```

The exit program printer file

This is the printer file used by the SWEEPEXIT exit program:

```
 * SWEEPPRTF from IFSSRC in RPGISCOOL                                           18
A          R SWEEPH                    SKIPB(1) SPACEA(2)
A                                    50'Process SubTree results'
A                                   120DATE(*JOB) EDTCDE(Y)
A                                   120TIME SPACEB(1)
A          R SWEEPD                    SPACEB(1)
A            TEXT         100         1
A            OBJECTPRT    132         1SPACEB(1)
A          R SWEEPE                    SPACEB(2)
A            TEXT         100         1
```

The user created command SWEEPC

The filename for the command source file is SWEEPC:

```
/* SWEEPC from IFSSRC in RPGISCOOL
/* Command definition for MERGESTMF program */
        CMD        PROMPT('Process subtree')
        PARM       KWD(PATHNAME) TYPE(*PNAME) LEN(100) +
                     PROMPT('Starting path name' 1)
        PARM       KWD(SUBTREE) TYPE(*LGL) RSTD(*YES) DFT(*YES) +
                     SPCVAL((*YES '0') (*NO '1')) +
                     PROMPT('Process Subdirectories' 2)
        PARM       KWD(OBJTYPES) TYPE(*CHAR) LEN(10) DFT(*ALL) SNGVAL((*ALL)) +
                     MIN(0) MAX(66) PGM(*YES) CHOICE('Object Type, *ALL') +
                     PROMPT('Object Types' 3)
        PARM       KWD(LOCALRMT) TYPE(*CHAR) LEN(1) RSTD(*YES) DFT(*ALL) +
                     VALUES(0 1 2) SPCVAL((*ALL '0') (*LCL '1') (*RMT '2')) +
                     CHOICE('*ALL, *LCL, *RMT') +
                     PROMPT('Local or Remote Objects' 4)
        PARM       KWD(INEXCLUTY) TYPE(*CHAR) LEN(1) RSTD(*YES) DFT(*NONE) +
                     VALUES(' ' 0 1) SPCVAL((*NONE ' ') (*INCL '0') (*EXCL '1')) +
                     CHOICE('*NONE, *INCL, *EXCL') PROMPT('IN_EXclusion type' 5)
        PARM       KWD(INEXCLUPN) TYPE(*PNAME) LEN(100) MIN(0) MAX(10) +
                      PMTCTL(INEXCL) PROMPT('IN_EXclusion Directory' 10)
        PARM       KWD(ERRORRCV) TYPE(*CHAR) LEN(1) RSTD(*YES) +
                     DFT(0) VALUES(0 1 2 3 4) MIN(0) +
                     PROMPT('ErrorRecovery Action' 6)
        PARM       KWD(USERTEXT) TYPE(*CHAR) LEN(100) +
                     PROMPT('User Function Text' 7)
INEXCL: PMTCTL     CTL(INEXCLUTY) COND((*NE ' '))
        DEP        CTL(&PATHNAME *NE ' ') PARM((&INEXCLUTY *NE '0'))
        DEP        CTL(&PATHNAME *EQ ' ') PARM((&INEXCLUTY *EQ '0'))
```

Main module prototype

The member name for the main module prototype used for Input parameters definition is SWEEPPR:

```
 * SWEEPPR from IFSSRC in RPGISCOOL
 * Prototype for Sweep program                                              1
D Sweep           PR                  Extpgm('SWEEP')
D  InputPath                   100A
D  SubtreeParm                   1S 0
D  ObjTypes                    662A
D  LocalRemote                   1A
D  In_ExclusionT                 1A
D  In_ExclusionP              1002A
D  ErrorRcv                      1A
D  FuncCtrlBlk                 100A
```

Path name structure definition

The member name for the path name structure definition is PATHNAMEDF. The complete source code can be found in 5.7.3.1, “Path name structure or path name format” on page 254.

Prototype for the Qp0lProcessSubtree API exit program

The member name for the Qp0lProcessSubtree API and the associated exit program "Process a path name" is PRCSUBTRPR. The API prototype source code can be found in 5.7.5.2, “The Qp0lProcessSubtree() API” on page 282. The exit program prototype source code can be found in 5.7.5.3, “Process a path name exit program” on page 290. Both prototypes should be merged into the same member.

The Qp0lProcessSubtree structure definition

The member name for the Qp0lProcessSubtree structure definition is PRCSUBTRDF. This member should include the structure definition of three pointer parameters used to call the API. The three definitions are:

Array of Object type:	Described in “Array of object types” on page 285.
In_Exclusion list:	Described in “IN_Exclusion list” on page 286.
User Function pointer:	Described in “User function pointer” on page 288.

Prototype for the DspError subprocedure

The member name for the DspError subprocedure, used to display error messages and to call C runtime function related to error handling, is ERRPROTO. The complete source code of prototypes can be found in “Prototype for the DspError subprocedure” on page 301.

DspError subprocedure

The DSPError module is used to display error messages that occur during API processing. The subprocedure retrieves the content of the errno variable and retrieves the associated text. Both information, plus a text location input parameter, is displayed on the screen. This external module needs to be bound at program creation time with the main module. More information on the DspError subprocedure can be found in 5.7.6.2, “The DspError subprocedure” on page 301.

Try it yourself

The following commands can be used to create the sample application on your system:

```
CRTRPGMOD MODULE(RPGISCOOL/SWEEP) SRCFILE(RPGISCOOL/SWEEPSRC)
CRTPRTF FILE(RPGISCOOL/SWEEPPRTF) SRCFILE(RPGISCOOL/SWEEPSRC)
CRTRPGMOD MODULE(RPGISCOOL/SWEEPEXIT) SRCFILE(RPGISCOOL/SWEEPSRC)
CRTPGM PGM(RPGISCOOL/SWEEP) MODULE(RPGISCOOL/SWEEP RPGISCOOL/SWEEPEXIT
 RPGISCOOL/DSPERROR)
```

Do not forget to also create the DspError module, discussed in “The DspError subprocedure” on page 302, which is used to display error messages (described in “DspError subprocedure” on page 300).

Also, the IFS APIs and the C runtime functions are included in a binding directory named QC2LE. We chose to specify the binding directory (BDNDIR) on the H-Spec of our modules. Instead, the binding directory can be explicitly specified on the CRTPGM command.

You can use the user created command as described in 5.7.5.1, “Overall concept of the sample program SWEEP” on page 280, to call the program.

5.7.6 IFS APIs error reporting

The UNIX-type APIs described in *System API Reference OS/400 UNIX-Type APIs,* SC41-5875, support APIs that use *errno* to report error conditions. This section defines how to retrieve these errors when using UNIX-type APIs in RPG IV.

5.7.6.1 Description of the errno value

The UNIX-type APIs, as described earlier, are mostly C functions. All these functions include the <errno.h> include file in their code. This include file defines macros that are set to the errno variable. The <errno.h> include file defines macros for values that are used for error reporting in the C library functions and defines the macro errno. An integer value can be assigned to errno, and its value can be tested during run time.

When an error occurs during the API processing, these C functions set errno to specific values, depending on the type of error.

Your program can use the __errno() and the strerror() to retrieve the error value and message text to display or print them. The strerror() function returns a pointer to an error message string that is associated with errno. The __errno() and strerror() functions should be used immediately after a function is called since subsequent calls may alter the errno value.

The list of the errno values that can be set by a specific API are usually described in the API documentation. The list of all the *errno* values can be found in Chapter 14, "Errno Values for UNIX-Type Functions" in the *System API Reference OS/400 UNIX-Type APIs*, SC41-5875.

ILE C Programmer's Guide, SC09-2712, contains more information on the errno. This manual suggest that you use the perror() function to print the error number value. We decided to use the DSPLY operation code, instead, since the perror brings a C interface (strerr), which is not compliant with the RPG style.

5.7.6.2 The DspError subprocedure

In the two previous sample applications, a subprocedure named DspError was used to retrieve the error number created when a problem occurred with the IFS APIs processing and to retrieve the message text associated with this error number. The subprocedure displays text as the input parameter, the error number, and the first 37 bytes of the associated message text.

We used the following two C runtime functions in the subprocedure to handle the errno value set by the IFS APIs. Both of these are described in the *ILE C for AS/400 Run-Time Library Reference,* SC41-5607.

__errno()	Values for IFS Enabled C Stream I/O
strerror()	Set Pointer to Runtime Error Message C function

Prototype for the DspError subprocedure

Here is the prototype used for the input parameter of the subprocedure:

```
 * ERRPROTO from IFSSRC in RPGISCOOL
 * Prototype for DspError subprocedure
DDspError         PR                  Extproc('DspError')
D text                           10A   Const
```

One parameter is received by the subprocedure. The 10-byte *text* variable is displayed on the screen and is used to indicate a marker in the calling program where the error occurred.

Prototypes for the error handling APIs

Here are the prototypes for the two APIs used in our sample program:

```
 * ERRPROTO from IFSSRC in RPGISCOOL
 * Prototype for __errno() and strerror()
 *    From the ILE C for AS/400 Run-Time Library Reference
DGetErrNo         PR              *   Extproc('__errno')

DStrError         PR              *   ExtProc('strerror')
D errorNo                      10I 0 Value
```

The __errno() function returns the memory location of the value of the errno field. The strerror() function uses this value to retrieve the memory location of the message text associated with this error number.

The DspError subprocedure

Here is the sample code of the DspError subprocedure. This subprocedure needs to be bound with the module using it.

```
 * DSPERROR from IFSSRC in RPGISCOOL
 ****************************************************************
 *Subprocedure: DspError
 *
 * Uses to display the error nbr, the error text and a location
 *  parameter using the DSPLY (display) op-code
 ****************************************************************

H nomain BNDDIR('QC2LE')

 /Copy RPGISCOOL/IFSSRC,ERRPROTO

PDspError         B                   Export
DDspError         PI
D text                          10A   Const

 * Variables for __errno() and strerror() APIs
DerrorNo@         S               *   Inz
D errorNo         S             10I 0 Based(Errorno@)
DerrorMsg@        S               *   Inz
D errorMsg        S            100A   Based(errorMsg@)
Derrortxt         S             52A   Inz(*blank)

C                   Eval      errorNo@ = GetErrNo
C                   Eval      errorMsg@ = StrError(errorNo)
C                   Eval      errortxt =%trim(text) + '->' +
C                                         %char(errorNo)
C                             + ':' + %subst(errormsg : 1 : 37)
C     errortxt      Dsply
P                 E
```

DSPError module creation

You can use the following commands to create the DSPError module, which contains the DSPError external subprocedure:

```
CRTRPGMOD MODULE(RPGISCOOL/DSPERROR) SRCFILE(IFSSRC)
```

Don't forget to bind this module with the main module on the program object creation. Also, the IFS APIs and the C runtime functions are included in a binding directory named QC2LE. We chose to specify the binding directory (BDNDIR) on the H-Spec of our modules. The binding directory can be explicitly specified on the CRTPGM command.

5.7.7 More information about IFS APIs in RPG IV

Multiple articles and coding examples have been written on using the IFS APIs in RPG, and most of them are available on the Internet. Here is a list of a few of them:

- "RPG and COBOL integrated file system code examples" can be found on the AS/400 Information Center found at: `http://www.as400.ibm.com/infocenter`

  Once you reach the Information Center, select **Database and File Systems**, and then **Integrated file system**. You should find **Example: RPG code snippets** under **Examples**.
- "Adding High-Level Math to RPG IV, Interfacing with the C Language is easy with prototypes" can be found on the RPG Developer Network News on RPGIV.com at: `http://www.rpgIV.com/rpgnews/Feb99a/highmath.html`
- "Stealing Time, The Easy Way to Format a Date or Time Value" can be found on the RPG Developer Network News on RPGIV.com at: `http://www.rpgIV.com/rpgnews/Feb99b/timerpg.html`

5.8 User exit programs

This section discusses tips and techniques on using RPG programs as *user exit programs*. User exit programs are used in conjunction with OS/400 exit points in the System Registration Information. An FTP client/server request validation program is used as a programming example in this section.

5.8.1 The system registry

The registration facility provides a central point to store and retrieve information about OS/400 and non-OS/400 exit points and their associated exit programs. This information is stored in the registration facility repository and can be retrieved to determine which exit points and exit programs already exist.

You can use the registration facility APIs to register and de register exit points, to add and remove exit programs, and to retrieve information about exit points and exit programs. You can also perform some of these functions by using the Work with Registration Information (WRKREGINF) command.

An *exit point* is the point in a system function or program where control is turned over to one or more exit programs to perform a function. The exit point provider is responsible for defining the exit point information, defining the format in which the exit program receives data, and calling the exit program. Of course, these exit programs can be written in RPG IV!

5.8.2 The FTP client/server validation request exit points

The FTP client/server request validation exit program gives you control over whether an operation (an FTP subcommand) is allowed or rejected. Decisions made by exit programs are in addition to any validation performed by the FTP client/server application. The FTP client/server request validation exit program is called each time one of these requests are processed within an FTP session:

- Session initialization
- Directory/library creation
- Directory/library deletion
- Setting current directory
- Listing file names
- File deletion
- Sending a file
- Receiving a file
- Renaming a file
- Executing a CL command on the FTP server

The exit point used in our example for this function is the FTP server request validation QIBM_QTMF_SERVER_REQ. The parameter format VLRQ0100 is used by this exit point to pass parameters to the exit program.

Four different TCP/IP applications share the VLRQ0100 interface through their own exit points. Our sample program can be used without modifications for all of them. The first parameter identifies which application is calling the exit point, and therefore the exit program. These application identifiers are:

- FTP client program (QIBM_QTMF_CLIENT_REQ)
- FTP server program (QIBM_QTMF_SERVER_REQ)
- REXEC server program (QIBM_QTMX_SERVER_REQ)
- TFTP server program (QIBM_QTOD_SERVER_REQ)

Table 76 contains the required parameter format for the VLRQ0100 exit point interface.

Table 76. Required parameter format for the VLRQ0100 exit point interface

Parameter	Description	Usage	Type and length
1	Application identifier	Input	Integer(10)
2	Operation identifier	Input	Integer(10)
3	User profile	Input	Char(10)
4	Remote IP address	Input	Char(10)
5	Length of remote IP Address	Input	Integer(10)
6	Operation-specific information	Input	Char(*)
7	Length of operation-specific information	Input	Integer(10)
8	Allow operation	Output	Integer(10)

The *Allow operation* parameter is considered as the return value passed from the exit program to the exit point. The value of this parameter indicates that the FTP server should reject or allow the requested operation.

The parameter descriptions can be found in the exit point documentation, available on the AS/400 Information Center found at:
`http://www.as400.ibm.com/infocenter`

Once you reach the Information Center site, select **TCP/IP**, and then **Transferring files (FTP)** and **FTP security controls**. The Application Identifier, Operation Identifier, and Allow Operation (return value) parameters are defined in the comments of our sample programs.

5.8.3 The FTP client/server request validation sample program

The client/server validation request exit program receives the parameters described by the VLRQ0100 exit point interface for any of the exit points using this format. This program uses the accounting code value, stored in the user profile information, to validate the usage of the requester operation. The accounting code is a 15-byte field which can be viewed, specified, or modified by the regular OS/400 commands or APIs related to user profile, such as the Create User Profile (CRTUSRPRF) and Change User Profile (CHGUSRPRF) commands.

The sample program can support all the application identifiers stated previously, using the same program at multiple exit points. Based on this support, this program extensively uses the expression format available in RPG IV.

The format of the accounting code used by this program is described in Table 77.

Table 77. Format of the accounting code parameter

VLRQ0100 parameter	Parameter values	Account code index	Access values
Application ID	0 (FTP Client request)	1	0 = Never Allowed 1 = Allowed 9 = Always Allowed
	1 (FTP Server request)	2	
	2 (Rexec Server request)	3	
	3 (Tftp Server request)	4	
Operation ID	0 (Initialize Session)	6	0= Not Allowed, All dir/lib 1= Only Client, All dir/lib 2= Only Server, All dir/lib 3= Only Rexec, All dir/lib 4= Only TFTP, All dir/lib 5= Only Client, Public dir/lib 6= Only Server, Public dir/lib 7= Only Rexec, Public dir/lib 8= Only TFTP, Public dir/lib 9= All functions, All dir/lib A= All functions,Pub dir/lib B= Client, All and Svr, Pub C= Client, Pub and Svr, All ...
	1 (Create Directory/Library)	7	
	2 (Delete Directory/Library)	8	
	3 (Set Current Library)	9	
	4 (List Directory/Library)	10	
	5 (Delete Files)	11	
	6 (Send Files)	12	
	7 (Receive Files)	13	
	8 (Rename Files)	14	
	9 (Execute CL command)	15	

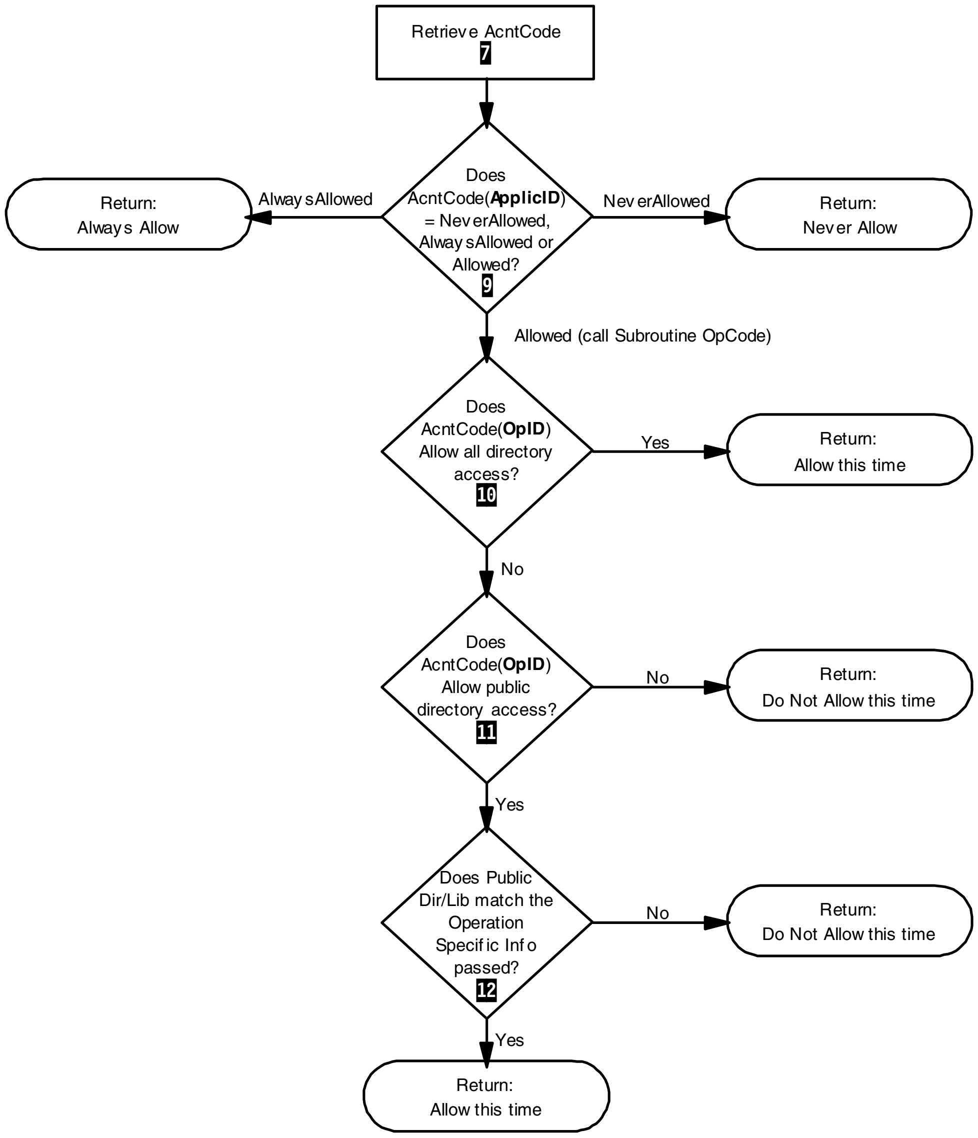
As you can see in Table 77, many additional Operation ID access value combinations can be added. In the sample program (see “FTPRQSEXIT code sample” on page 307), the possible values of the parameters passed by the exit point are shown with marker 2. The possible values available for the access mode in the accounting code are shown with marker 4.

The sample program uses the Retrieve User Information (QSYRUSRI) API, hidden in the subprocedure *RtvUsrPrf*, to retrieve the accounting code from the user profile ID passed as a parameter (see marker 8). For default users as anonymous or QTCP, the default accounting code value has been coded directly into the program (please see marker 5). The anonymous user ID is an FTP feature that doesn’t require an ANONYMOUS user profile to exist on the system. Since QTCP is a system user profile, we prefer not to modify the accounting code on the user profile itself (user profile QTCP is used for any server request before the FTP logon).

This program example is based on an example available on the AS/400 Information Center found at: `http://www.as400.ibm.com/infocenter`

Once you reach the Information Center site, select **TCP/IP**, and then **Transferring files (FTP)** and **FTP security controls**. We decided to keep the public directory and library validation, which are specified as a constant in the definition specification (see marker 12). We know there may be a better way of executing this validation, such as scanning the string for specific characters that identify the type of directory instead of relying on the length of the field, but we wanted to keep this part of the program as simple as possible.

Figure 34 on page 307 describes the outline of our sample program.



Retrieve AcntCode
7
Does AcntCode(**ApplicID**) = NeverAllowed, AlwaysAllowed or Allowed?
9
AlwaysAllowed
Return: Always Allow
NeverAllowed
Return: Never Allow
Allowed (call Subroutine OpCode)
Does AcntCode(**OpID**) Allow all directory access?
10
Yes
Return: Allow this time
No
Does AcntCode(**OpID**) Allow public directory access?
11
No
Return: Do Not Allow this time
Yes
Does Public Dir/Lib match the Operation Specific Info passed?
12
No
Return: Do Not Allow this time
Yes
Return: Allow this time

Figure 34. FTP client/server validation request exit program flowchart

5.8.3.1 FTPRQSEXIT code sample

Here is the sample code. Refer to the program note for more explanation on the code, using the markers as a reference:

```
 * FTPRQSEXIT from EXITSRC in RPGISCOOL
 * Program: FTPRQSEXIT   FTP Client/Server Request Exit program
 *
 * This program can be use to restrict FTP functions on the AS/400.
 * This program needs to be register in the following exit point:
 *  QIBM_QTMF_SERVER_REQ and QIBM_QTMF_CLIENT_REQ
 *  of the registration facility (WRKREGINF). It also receives
 *  its parameters from those exit points as described in the
 *  TCP/IP Configuration and Reference SC41-5420 manual.

H Option(*SrcStmt)

 * Prototype and Procedure Interface for Input Parameters        1
 /COPY RPGISCOOL/EXITSRC,FTPRQSPROT

D FtpRqsExit      PI
```

```
D  ApplicID                     10I 0 CONST
D  OpID                         10I 0 CONST
D  UsrPrf                       10A   CONST
D  RmtIPAddr                    10A   CONST
D  RmtIPAddrL                   10I 0 CONST
D  OpSpecInf                   999A   CONST
D  OpSpecInfL                   10I 0 CONST
D  RV                           10I 0

 * Application ID values:                                         2
D FTPClientRqs    C                   0
D FTPServerRqs    C                   1
D RexecRqs        C                   2
D TftpRqs         C                   3
 * Operation ID values:
D InitSession     C                   0
D CreateDirLib    C                   1
D DeleteDirLib    C                   2
D SetCurrentLib   C                   3
D ListDirLib      C                   4
D DeleteFiles     C                   5
D SendFiles       C                   6
D ReceiveFiles    C                   7
D RenameFiles     C                   8
D ExecuteCL       C                   9
 * Return Value (RV) possible values
D NeverAllow      C                   -1
D DontAllow       C                   0
D Allow           C                   1
D AllowAllw       C                   2

 * Prototype for Retrieve User Information (QSYRUSRI) API          3
 /COPY RPGISCOOL/EXITSRC,QSYRUSRIPR
 * Parameters required
D RcvVar          DS
D AcntCodeC             310    324A
D AcntCode              310    324A   Dim(15)
D RcvVarL         S             10I 0 Inz(324)
D FmtName         S              8A   Inz('USRI0300')
 * Include Error Code Parameter structure definition
 /COPY QSYSINC/QRPGLESRC,QUSEC

 * Structure of the accounting code index:                        4
 * 1 = Client Request Allowed
 * 2 = Server Request Allowed
 * 3 = REXEC Request Allowed
 * 4 = TFTP Request Allowed
 *  possibles values:
D NeverAllowed    C                   '0'
D Allowed         C                   '1'
D AlwaysAllowed   C                   '9'
 * 5 = Reserved
 * 6 to 15 = (mapping the OpID index value (0 to 9))
 *            All = All directories or libraries
 *            Pub = Public directory or library (see pgm constants)
D AllNotAllowed   C                   '0'
D OnlyClientAll   C                   '1'
D OnlyServerAll   C                   '2'
D OnlyRexecAll    C                   '3'
D OnlyTftpAll     C                   '4'
D OnlyClientPub   C                   '5'
D OnlyServerPub   C                   '6'
D OnlyRexecPub    C                   '7'
D OnlyTftpPub     C                   '8'
D AnyAll          C                   '9'
D AnyPub          C                   'A'
D CltAllSvrPub    C                   'B'
D CltPubSvrAll    C                   'C'

 * Program constants                                              5
D anonymous       C                   'ANONYMOUS '
D anonymousAC     C                   '01xx20066060000'
D QTCP            C                   'QTCP      '
D QtcpAC          C                   '11xx90000000000'
D PublicLib       C                   '/QSYS.LIB/ITSOIC400.LIB'
D PublicDir       C                   '//ITSOIC.400'

 * Prototype for xlate subprocedure                                 6
 /COPY RPGISCOOL/EXITSRC,XLATEPROT

 * Set accouting code value                                        7
C                   Select
 * If user profile is anonymous or QTCP, set default
 *  accounting code instead.
C                   When      UsrPrf = Anonymous
C                   Eval      AcntcodeC = AnonymousAC
C                   When      UsrPrf = QTCP
C                   Eval      AcntCodeC = QtcpAC
C                   Other
 * Othwerwise, Retrieve the accounting code from                   8
 *  user profile using QSYRUSRI API
C                   Eval      QUSBPRV = 0
C                   CallP     RtvUsrPrf (RcvVar : RcvVarL :
C                                        FmtName : UsrPrf : QUSEC)
C                   EndSl

 * Lookup general access in Accounting code using Application ID   9
 * value as offset
C                   Select
C                   When      AcntCode(ApplicID + 1) = NeverAllowed
C                   Eval      RV = NeverAllow

C                   When      AcntCode(ApplicID + 1) = AlwaysAllowed
C                   Eval      RV = AllowAllw

C                   When      AcntCode(ApplicID + 1) = Allowed
 * Execute OpCode subroutine to validate specific operation on all
 *  or public directories
C                   Exsr      OpCode

C                   Other
C                   Eval      RV = DontAllow
C                   EndSl

 * End of program
C                   Eval      *inLR = *ON
C                   Return

 *---------------------------------------------------------------
 * opCode Subroutine

C     opCode        Begsr

 * Lookup operation access in acounting code using the
 *  Operation ID as offset
C                   Select
 * If all directory access, Allow operations                        10
C                   When        AcntCode(OpID+6) =
C                                           %char((ApplicID + 1))
C                               or AcntCode(OpID+6) = AnyAll
C                               or ApplicID = FTPClientRqs
C                                and Acntcode(OpID+6) = CltAllSvrPub
C                               or ApplicID = FTPServerRqs
C                                and Acntcode(OpID+6) = CltPubSvrAll
C                   Eval      RV = Allow
 * If public directory operation, allow only if public directory   11
 *  or lib is used
C                   When       AcntCode(OpID+6) =
C                                            %char((ApplicID + 5))
C                               or AcntCode(OpID+6) = AnyPub
C                               or ApplicID = FTPServerRQS
C                                 and Acntcode(OpID+6) = CltAllSvrPub
C                               or ApplicID = FTPClientRqs
C                                 and Acntcode(OpID+6) = CltPubSvrAll

 * Compare Public library and directory against constant           12
 *  use xlate subprocedure to translate directory to uppercase      6
C                   If        PublicLib = %Subst(OpSpecInf : 1 : 11)
C                               or PublicDir =
C                            xlate(%Subst(OpSpecInf : 1 : 23) : 'UP')
C                               or OpID = InitSession
C                               or OpID = ExecuteCl
 *                           (InitSession or ExecuteCL doesn't pass a
 *                              directory or library in OpSpecInf)
C                   Eval      RV = Allow
C                   Else
```

```
C                   Eval      RV = DontAllow
C                   EndIf

 * Operation not allowed
C                   Other
C                   Eval      RV = DontAllow
C                   EndSl

C                   EndSr

 *----------------------------------------------------------
 * xlate Subroutine                                          6

Pxlate            B
 * Procedure Interface definition
DXlate            PI           100A
D InputString                  100A   Value
D Direction                      1A   Value
 * Direction values : U = from lower to upper
 *                    D = from upper to lower

 * Stand Alone variables
D OutputString    S            100A
D LW              C                   'abcdefghijklmnopqrstuvwxyz'
D UP              C                   'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

 * Translate Up or Down
C                   Select
C                   When      Direction = 'UP'
C     Lw:Up         Xlate     InputString   OutputString
C                   When      Direction = 'DN'
C     Up:Lw         Xlate     InputString   OutputString
C                   EndSl

 * Return OutputString value
C                   Return    OutputString
P                 E
```

FTPRQSEXIT program notes

1 As you can see, this program uses a procedure interface definition to define the parameters passed by the exit point. Here is the prototype definition that goes along with it:

```
 * FTPRQSPROT from EXITSRC in RPGISCOOL
 * Prototype for exit program FTPRQSEXIT
 *  using system registry format VLRQ0100
D FtpRqsExit      PR                  EXTPGM('FTPRQSEXIT')
D  ApplicID                     10I 0 CONST
D  OpID                         10I 0 CONST
D  UsrPrf                       10A   CONST
D  RmtIPAddr                    10A   CONST
D  RmtIPAddrL                   10I 0 CONST
D  OpSpecInf                   999A   CONST
D  OpSpecInfL                   10I 0 CONST
D  RV                           10I 0
```

6 The program uses a subprocedure to perform the translation of the directory value from lower case to upper case. The main benefit of using a subprocedure in this case is the use of the XLATE instruction, as a built-in function in an expression. Here is the prototype used for the XLATE procedure:

```
 * XLATEPROT from EXITSRC in RPGISCOOL
 * Prototype for xlate subprocedure
D Xlate           PR           100A
D  InputString                 100A   value
D  Direction                     1A   value
```

8 The Retrieve User Information (QSYRUSRI) API is called by using a CALLP instruction which requires a prototype:

```
 * QSYRUSRIPR from EXITSRC in RPGISCOOL
 * Prototype for Retrieve User Information (QSYRUSRI) API       3
D RtvUsrPrf       PR                  EXTPGM('QSYRUSRI')
```

```
D  RcvVar                        324A
D  RcvVarL                        10I 0 CONST
D  FmtName                         8A   CONST
D  UsrPrf                         10A   CONST
D  QUSEC                          16A
```

More information on this API can be found in the *System API Reference OS/400 Security APIs*, SC41-5872, manual. The usage of this OPM requires the definition of the error code structure definition QUSEC. This structure definition is imported from the library QSYSINC (QRPGLESRC/QUSEC) from the System Openness Include licensed program. For more information on using OPM APIs, refer to the 5.2, “Data queue APIs” on page 136.

The accounting code is referred in the program by looking in the mapping array definition AcntCode, which is a 15-element array of one character each.

9 The general access validation on the application ID is done by using the first four positions in the accounting code. Each position represents a specific application ID. The +1 offset is used as the first application ID number, which is 0. "FTP Client Request" matches the first index position in the accounting code array, which starts at 1.

10 The operation ID access validation is done by using the last 10 positions in the accounting code, starting at index number 6. The operation ID, passed as a parameter, is used as an offset to determine the exact location. Since the basic access value of any operation ID match the application ID sequence (value 1 to 4 against Application ID 0 to 3), we used the offset 1, plus the application ID, to validate some access values possibilities for a specific application and all access to directories or libraries.

11 The offset 5, plus the Application ID, is used to validate some access values possibilities for specific applications and public directory or library access. Please check program note **10** for more information.

Try it yourself

If you want to recreate the previous example, you can use the following commands. The /COPY members defined in the program notes **1**, **3**, and **6** need to exist in a source file prior to creating the module and program.

```
CRTRPGMOD MODULE(rpgiscool/ftprqsexit) SRCFILE(rpgiscool/exitsrc)
CRTPGM PGM(rpgiscool/ftprqsexit) MODULE(rpgiscool/ftprqsexit)
```

Please refer to the following section for an example on how to add the exit program to the FTP server validation request exit point in the system registry, or enter the following command:

```
ADDEXITPGM EXITPNT(QIBM_QTMF_SERVER_REQ) FORMAT(VLRQ0100) PGMNBR(1)
 PGM(RPGISCOOL/FTPRQSEXIT)
```

5.8.3.2 Registering the exit program

As stated earlier, this program can be used against all exit points by using the VLRQ0100 format. For example, to register the exit program to the FTP server Validation Request exit point in the system registry, perform the following steps:

1. Type `WRKREGINF` on a command line, and press Enter. A list of exit points appears, as shown in Figure 35.

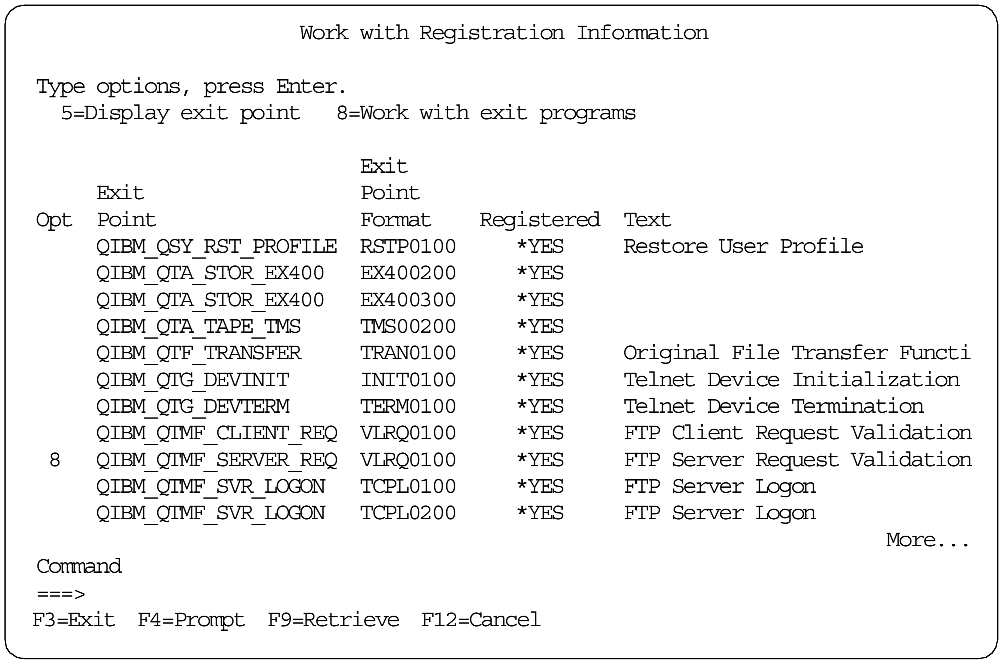

```
                         Work with Registration Information

 Type options, press Enter.
   5=Display exit point   8=Work with exit programs

                             Exit
      Exit                   Point
 Opt  Point                  Format    Registered  Text
      QIBM_QSY_RST_PROFILE   RSTP0100     *YES     Restore User Profile
      QIBM_QTA_STOR_EX400    EX400200     *YES
      QIBM_QTA_STOR_EX400    EX400300     *YES
      QIBM_QTA_TAPE_TMS      TMS00200     *YES
      QIBM_QTF_TRANSFER      TRAN0100     *YES     Original File Transfer Functi
      QIBM_QTG_DEVINIT       INIT0100     *YES     Telnet Device Initialization
      QIBM_QTG_DEVTERM       TERM0100     *YES     Telnet Device Termination
      QIBM_QTMF_CLIENT_REQ   VLRQ0100     *YES     FTP Client Request Validation
  8   QIBM_QTMF_SERVER_REQ   VLRQ0100     *YES     FTP Server Request Validation
      QIBM_QTMF_SVR_LOGON    TCPL0100     *YES     FTP Server Logon
      QIBM_QTMF_SVR_LOGON    TCPL0200     *YES     FTP Server Logon
                                                                        More...
 Command
 ===>
F3=Exit  F4=Prompt  F9=Retrieve  F12=Cancel
```

Figure 35. Work with Registration Information

2. Type 8 (Work with exit programs) next to the IBM_QTMF_SERVER_REQ Exit Point Format VLRQ0100 exit point as shown Figure 35, and press Enter. The Work with Exit Programs display appears as shown in Figure 36.

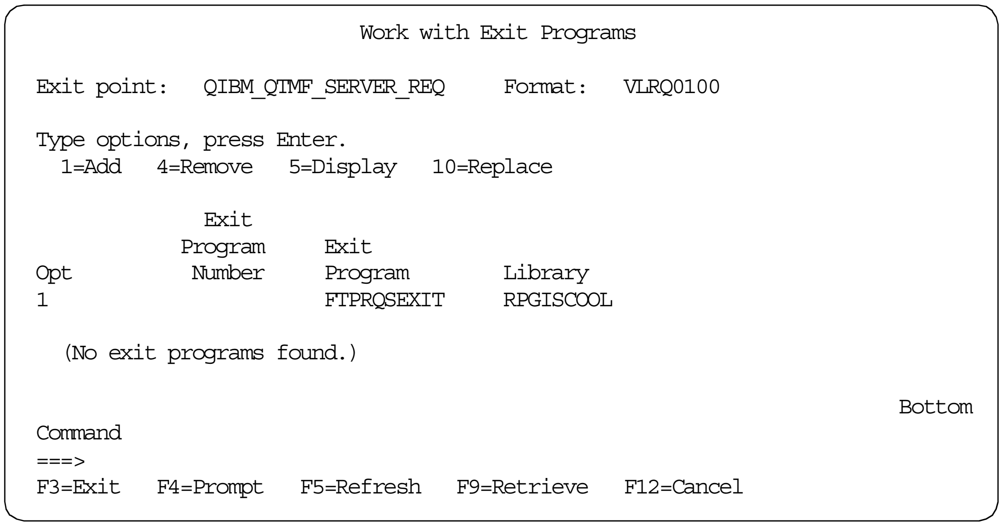
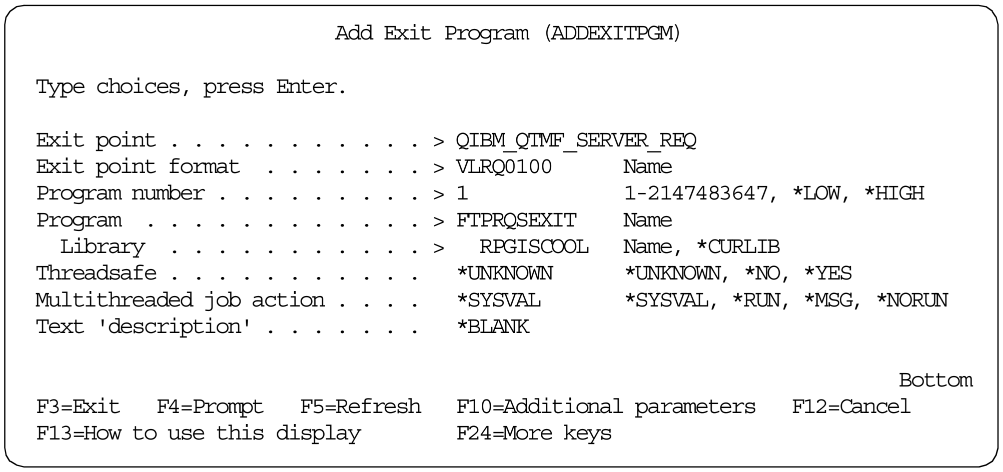

```
                              Work with Exit Programs

 Exit point:   QIBM_QTMF_SERVER_REQ     Format:   VLRQ0100

 Type options, press Enter.
   1=Add   4=Remove   5=Display   10=Replace

              Exit
            Program     Exit
 Opt         Number     Program        Library
 1                      FTPRQSEXIT     RPGISCOOL

   (No exit programs found.)

                                                                         Bottom
 Command
 ===>
 F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel
```

Figure 36. Work with Exit Programs display

3. Type 1 and the name of the compiled program as shown in Figure 36. Press Enter. The Add Exit Program (ADDEXITPGM) display appears, as shown in the Figure 37 on page 313.



```
                        Add Exit Program (ADDEXITPGM)

 Type choices, press Enter.

 Exit point . . . . . . . . . . . > QIBM_QTMF_SERVER_REQ
 Exit point format  . . . . . . . > VLRQ0100      Name
 Program number . . . . . . . . . > 1             1-2147483647, *LOW, *HIGH
 Program  . . . . . . . . . . . . > FTPRQSEXIT    Name
   Library  . . . . . . . . . . . >   RPGISCOOL   Name, *CURLIB
 Threadsafe . . . . . . . . . . .   *UNKNOWN      *UNKNOWN, *NO, *YES
 Multithreaded job action . . . .   *SYSVAL       *SYSVAL, *RUN, *MSG, *NORUN
 Text 'description' . . . . . . .   *BLANK

                                                                       Bottom
 F3=Exit   F4=Prompt   F5=Refresh   F10=Additional parameters   F12=Cancel
 F13=How to use this display        F24=More keys
```

Figure 37. Add Exit Programs display

4. Press Enter to process.

5.8.4 More information about user exit programs

Multiples articles and samples programs on using RPG IV as an exit program register on an exit point of the system registry are available on the Internet. You can also find a different FTP server validation request exit program and more information on this subject on the AS/400 Information Center Web site at:

`http://www.as400.ibm.com/infocenter`

Once inside the Information Center, select **TCP/IP** and then **Transferring files (FTP)** and **FTP security controls**.

Chapter 6. Database access with RPG IV

Probably one of the most powerful aspects of RPG IV on the AS/400 is its ability to integrate with the database. In this chapter, we review and provide 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 AS/400 system is no different. We use RPG IV as the common denominator as we explore the database access methods in the following sections:

- “Embedded SQL” on page 330
- “Stored procedures” on page 339
- “Call Level Interface” on page 348
- “Trigger programs” on page 379
- “Commitment control” on page 384

But first, we address the need to externalize your input and output (I/O) to allow your RPG IV applications to quickly adapt to new database requirements in 6.1, “Externalizing input and output” on page 315. After that, we show you how to replace the old RPG indicators with easy-to-read and built-in functions introduced in V4R2 in 6.2, “Replacing indicators with built-in functions” on page 327.

6.1 Externalizing input and output

Just what do we mean by the term *externalizing input/output (I/O)*? Aren’t our files externalized already? After all, we define them with DDS rather than use the old RPG II approach of defining files in input and output specifications.

In this section, we explain *externalizing I/O* and try to show you some of the benefits of taking this approach.

6.1.1 What we mean by externalizing

Externalizing means that all READ, CHAIN, and other database operations are located in separate routines and programs that require I/O make requests to these routines to perform the operation on their behalf. This section further explains this topic.

6.1.1.1 Why externalize?

Why would you want to do this? Perhaps we can best answer that question by posing one of our own examples.

Suppose that during discussions with your users, it becomes apparent to you that business needs have changed. After studying the new requirements for a while, you realize that you need to redesign the database to accommodate these changes.

Would you:

- 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 may be because your knowledge of your applications tells you that the database I/O is spread liberally throughout the programs. This makes the impact of a database change difficult to estimate and equally difficult to implement. Don't feel embarrassed if you pleaded "guilty". If you are entirely honest, this is the answer that 90% or more of all AS/400 RPG users would give.

Many of the applications that we 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 AS/400 system's older brother) 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 the same old problems). Of course it is easy to overlook the disadvantages when we're "under the gun" to produce as quickly as possible, and we can always re-do it later.

6.1.1.2 The benefits of externalizing

Externalizing I/O operations provides one way of helping to ensure that your applications can adapt quickly and (relatively) painlessly to changing business needs. Instead of coding a READ, CHAIN, etc. at each point in the program where database access is required, you invoke a routine to perform the I/O for you.

While many people have successfully implemented such schemes in RPG/400, doing so requires that you make program calls to the I/O routines. By utilizing RPG IV’s sub procedures, you can now provide this interface in a much more natural "English-like" fashion. For example, a routine to read a record from the Customer Master file may be called ReadCust and could be designed to either read a record by key or simply return the next record in the sequence. It may look something like this:

```
  * Read by key
 C                   Eval      CustRecord = ReadCust(CustKey)
  * Read next record
 C                   Eval      CustRecord = ReadCust
```

Alternatively, we may choose to use different names for the two functions, for example, ReadCustKey and ReadNextCust. If you need background information and details on the subprocedures, see Chapter 3, “Subprocedures” on page 31.

6.1.1.3 One step at a time

At first glance, it may seem that making these changes requires a lot of work. The main point to remember is that you don't 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 would be to switch each database to the new method the time you need to make changes to it.

6.1.2 Putting theory into practice: An example of externalizing I/O

In the following sections, we give you a brief, and admittedly highly simplified, example of using subprocedures to externalize your I/O. Hopefully by the time we finish you can see just how beneficial it can be. You may even have a few ideas about how you can apply the principles to your own applications.

The following sections lead you through this process:

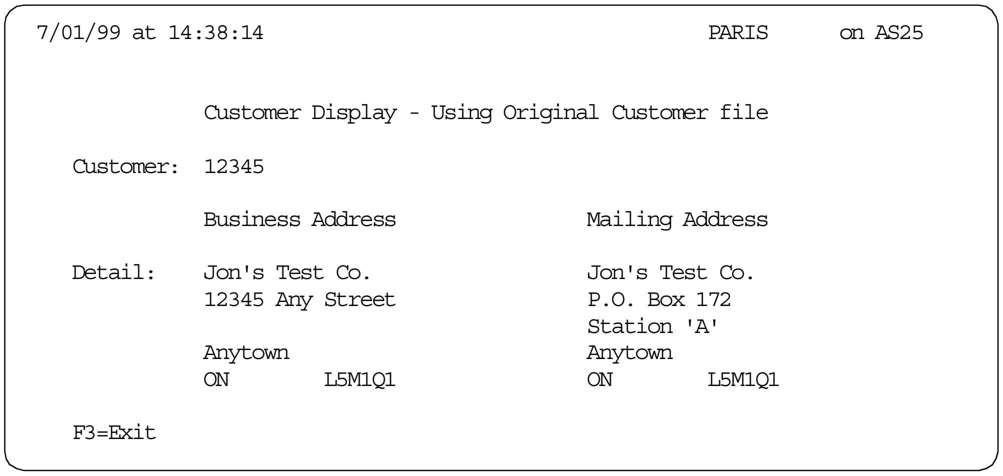
- We explain the *system* in its initial state. For the purpose of our example, the system consists of a single file and a single program. Needless to say, this is only intended to be representative of the overall system.
- The next section takes you through the steps required to move the file I/O operations into a separate subprocedure.
- The third section shows you how the changes in the underlying database design can be accommodated with little or no impact to the rest of the system.
- Finally, we discuss 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 that we will work with in this example.

6.1.3.1 The display file CUSTDISP

All of our programs in this section use the same display file. As you can see in Figure 38, it is very simple. We want to focus your attention on the real substance of the examples.



```
7/01/99 at 14:38:14                                  PARIS      on AS25


               Customer Display - Using Original Customer file

   Customer:   12345

               Business Address                 Mailing Address

   Detail:     Jon's Test Co.                   Jon's Test Co.
               12345 Any Street                 P.O. Box 172
                                                Station 'A'
               Anytown                          Anytown
               ON        L5M1Q1                 ON        L5M1Q1

   F3=Exit
```

Figure 38. The CUSTDISP display file

To use the program, you simply enter a five 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.

Note that the text `Original Customer file` that appears on the sample display is supplied by the program. Each of the samples display a different value.

Here is the source code for the 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  O  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        O 10 16REFFLD(CUSNAME)
A            B_ADDRESS1R        O 11 16REFFLD(ADDRESS1)
A            B_ADDRESS2R        O 12 16REFFLD(ADDRESS2)
A            B_CITY    R        O 13 16REFFLD(CITY)
A            B_PROVINCER        O 14 16REFFLD(PROVINCE)
A            B_POSTCODER        O 14 26REFFLD(POSTCODE)
A            M_CUSNAME R        O 10 48REFFLD(CUSNAME)
A            M_ADDRESS1R        O 11 48REFFLD(ADDRESS1)
A            M_ADDRESS2R        O 12 48REFFLD(ADDRESS2)
A            M_CITY    R        O 13 48REFFLD(CITY)
A            M_PROVINCER        O 14 48REFFLD(PROVINCE)
A            M_POSTCODER        O 14 58REFFLD(POSTCODE)
A                                 16  5'F3=Exit'
```

6.1.3.2 The initial database design

Our initial database doesn't really deserve the term "design". It is a System/36 style flat file with multiple record types. Sound familiar?

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.

This is the DDS for the file:

```
 *  File Name:           CUSTOMER
 *  File Description:    Customer Master File
 *
 *=============================================================
A                                     UNIQUE
A          R CUSTDET
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
```

6.1.3.3 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 attempt 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 appear in the appropriate address fields.

Here is the source. The comments relate to the numbered marks follow the source.

```
 * File SHOWCUST from EXTIOSRC in RPGISCOOL
FCustDisp  CF   E            WorkStn IndDs(D_Indicators)  1

FCustomer  IF   E           K Disk

D D_Indicators    DS                                      1
 * Response Indicators
D  Exit                   3      3N

 * Constants used in the program
D NotFound        C                   'Error - Customer not found'
D MailAddrErr     C                   'Error - Mail address missing'
D SameAsBus       C                   'Use business address'
D Business        C                   'B'
D Mailing         C                   'M'
D Separate        C                   'M'

 * External data structures used for database and display I/O   2
D CustAddr      E DS                  ExtName(Customer)
D BusAddr       E DS                  ExtName(Customer) Prefix(B_)
D MailAddr      E DS                  ExtName(Customer) Prefix(M_)

C                   Eval      Version = 'Original Customer file'

 * Initial 'priming' read of display file
C                   ExFmt     DispRec

C                   DoU       Exit

 * Set up key and search for Business address
 *   Note: Key list CustKey is defined at the end of the source
C                   Eval      AddrFlag = Business
C     CustKey       Chain     Customer

 * If the record is found set up display data and retrieve
 * Mailing address if MailFlag indicates that one is available
C                   If        %Found(Customer)
C                   Eval      BusAddr = CustAddr

C                   If        MailFlag <> Separate
C                   Eval      MailAddr = *Blanks
C                   Eval      M_CusName = SameAsBus
C                   Else
 * Set up key and retrieve mailing address record
C                   Eval      AddrFlag = Mailing
C     CustKey       Chain     Customer

C                   If        %Found(Customer)
C                   Eval      MailAddr = CustAddr
C                   Else
 * Error - no mailing record found - display error text
C                   Eval      MailAddr = *Blanks
C                   Eval      M_CusName = MailAddrErr
C                   EndIf
C                   EndIf

C                   Else
 * Requested Customer not found - display error text
C                   Eval      BusAddr = *Blanks
C                   Eval      MailAddr = *Blanks
C                   Eval      B_CusName = NotFound
```

```
 C                   EndIf
  * Display results and accept next request
 C                   ExFmt     DispRec

 C                   EndDo

 C                   Eval      *InLR = *On

 C     CustKey       Klist
 C                   KFld                    Cust#
 C                   KFld                    AddrFlag
```

1 This example takes advantage of the Indicator Data Structure (INDDS) facility added to RPG IV in V4R2. This allows the programmer to simply assign names to response and conditioning indicators. The From/To notation method has to be used to indicate which indicator is being defined. In our example, display file indicator 03 is associated with the name Exit.

Notice that 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) 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 are the names used to display the data. This 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

Create the display file from the source:

```
CRTDSPF FILE(RPGISCOOL/CUSTDISP) SRCFILE(RPGISCOOL/EXTIOSRC)
SRCMBR(CUSTDISP)
```

Create the customer physical file:

```
CRTDSPF FILE(RPGISCOOL/CUSTOMER) SRCFILE(RPGISCOOL/EXTIOSRC)
SRCMBR(CUSTOMER)
```

Here is a partial screen shot with a sample of the data 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
```

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

In this section, we cover the steps involved with separating the database logic from the display logic. This sets us up for the subsequent database changes and also provides us with database I/O routines that could be used, for example, from a Java client.

6.1.4.1 The new structure

We are going to create a new version of the main program SHOWCUST, to which we will give the highly original name of SHOWCUST2. For the purposes of our example, we decided to keep the display file I/O in this main program. It is so simple that there would be nothing left if we were to externalize that portion of the logic! However, in the real world, externalizing the display file I/O would also be a very useful since it would allow us to replace the 5250 screen with a Java client for example.

The database I/O functions will be moved to a new subprocedure GETCUST2. As you will see later, once the separation has been made, it becomes far easier to accommodate changes in the database design.

6.1.4.2 The GETCUST2 subprocedure

A brief study of this code reveals that the logic itself is almost unchanged from the original code in SHOWCUST.

Here is the source for the sub procedure GetCust. The markers 1 through 4 are defined immediately after this source code example.

```
 * GETCUST2 from EXTIOSRC in RPGISCOOL
H NoMain

FCustomer  IF   E           K Disk
 * Copy in the prototype for the subprocedure GetCust
 /Copy RPGisCool/ExtIOSrc,GetCustPr 1

 * Constants used in the program
D MailAddrErr     C                   'Error - Mail Address Missing'
D SameAsBus       C                   'For Mail Use Business Address'
D Business        C                   'B'
D Mailing         C                   'M'
D Separate        C                   'M'
D Found           C                   *On
D NotFound        C                   *Off

 * Data structure used when retrieving Customer records
D CustAddr      E DS                  ExtName(Customer)

 * Structures used to pass Business & Mailing address data to caller
D BusAddr       E DS                  Export            4
D                                     ExtName(Customer) Prefix(B_)

D MailAddr      E DS                  Export           4
D                                     ExtName(Customer) Prefix(M_)

 * Beginning of subprocedure GetCust
P GetCust         B                   Export       2
D                 PI              N
D  CustNum                            Like(Cust#)

 * Set up key and search for Business address
C                   Eval      AddrFlag = Business
C     CustKey       Chain     Customer

 * If the record is found set up data and retrieve
 * Mailing address if MailFlag indicates that one is available
C                   If        %Found(Customer)
C                   Eval      BusAddr = CustAddr

C                   If        MailFlag <> Separate
C                   Eval      MailAddr = *Blanks
C                   Eval      M_CusName = SameAsBus

C                   Else
 * Set up key and retrieve mailing address record
C                   Eval      AddrFlag = Mailing
C     CustKey       Chain     Customer

C                   If        %Found(Customer)
```

```
C                   Eval      MailAddr = CustAddr
C                   Else
 * Error - no mailing record found - set up error text
C                   Eval      MailAddr = *Blanks
C                   Eval      M_CusName = MailAddrErr

C                   EndIf
C                   EndIf
 * We have retrieved at least the Business address so return found
C                   Return    Found         3

C                   Else
 * Requested Customer not found - set up error text and return
C                   Eval      BusAddr = *Blanks
C                   Eval      MailAddr = *Blanks
C                   Eval      B_CusName = NotFound
C                   Return    NotFound         3
C                   EndIf

C     CustKey       Klist
C                   KFld                    CustNum
C                   KFld                    AddrFlag

P GetCust         E
```

1 First, we produced the prototype for the new subprocedure and added a /Copy statement to bring that source member into the program.

2 Next, we added the procedure interface. If you read Chapter 3, “Subprocedures” on page 31, this should all be familiar to you. If you jumped straight into this chapter, you may want to consider going back and reading this chapter now before continuing with this section since we do not describe these features here.

3 Next, we added the Return op-code. In our example, we chose to return a named indicator (data type N) to notify the caller of the success or failure of the read operation.

4 Since the purpose of the subprocedure is to retrieve the Customer information on behalf of its caller, we need to find some way to do this. There are a number of different methods that we could have used, but we chose to use ILE’s Import/Export capability. The data read from the file is "exported" so that it can be "imported" by the main program. This allows the two procedures to share the data without the need for it to be passed as parameters. Some programmers do not like this approach, but in this limited context, it seems to be quite the method to use. Later in the chapter, we’ll discuss alternative methods that could have been used.

In our example, the Import/Export data structures use the definitions of the underlying databases. This was done to simplify the example. In practice, you may choose to develop a separate composite definition for the express purpose of passing back the result set.

6.1.4.3 The main program SHOWCUST2

Program SHOWCUST2 is a modified version of the original SHOWCUST. Here is the source for SHOWCUST2:

```
H DftActGrp(*No)

FCustDisp  CF   E             WorkStn IndDs(D_Indicators)

 /Copy GetCustPr       2

D D_Indicators    DS

 * Response Indicators
D  Exit                   3      3N
```

```
 * Constants used in the program
D NotFound        C                   'Error - Customer not found'

D BusAddr       E DS                  Import 3
D                                     ExtName(Customer) Prefix(B_)

D MailAddr      E DS                  Import 3
D                                     ExtName(Customer) Prefix(M_)

C                   Eval      Version = 'External I/O - Version 1'

C                   ExFmt     DispRec

C                   DoU       Exit

 * GetCust reurns an indicator which is on if no record was found
C                   If        Not GetCust(Cust#)      1
C                   Eval      B_CusName = NotFound
C                   EndIf

C                   ExFmt     DispRec

C                   EndDo

C                   Eval      *InLR = *On
```

1 The first modification made was to remove the file I/O logic and replace it with an invocation of the GetCust sub procedure.

2 The next step was the addition of a /COPY statement to incorporate the prototype for GetCust that we 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. Now whenever GetCust places data into these structures, it is "visible" to our program.

Try it yourself

To recreate this example on your system, you need to compile the subprocedure and the main program using the following commands:

Note: You can create a service program from GETCUST2 and bind that to the SHOWCUST2 module, but simply 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

Let us imagine that we have decided that the time has come to perform additional normalization on the database. We 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.

6.1.5.1 Database changes

For the sake of simplicity, the two new databases will retain the original field names. RPG IV's PREFIX keyword gives us all the power we need to simply rename the fields at the program level. Each of the new files is keyed only on the Customer number (CUST#).

As you can see in the following example, the source for the CUSTOMERB file is almost identical to the original CUSTOMER file (see 6.1.3.2, "The initial database design" on page 318) with the exception that the record format was renamed to CUSTDETB.

```
  *  CUSTOMERB from EXTIOSRC in RPGISCOOL
  *
  *  File Name:           CUSTOMERB
  *  File Description:    Customer Business Addresses
  *
  *===============================================================
 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 but the "Separate mailing address" flag has been removed and its record format changed to CUSTDETM. Here is the source:

```
  *  CUSTOMERM from EXTIOSRC in RPGISCOOL
  *
  *  File Name:           CUSTOMERM
  *  File Description:    Customer Mailing Addresses
  *
  *============================================================
 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

Source GETCUST3 contains a modified version of the original subprocedure GetCust. It now retrieves the customer data by accessing the two separate databases. Here's the modified source. An explanation of the marked items follows.

```
  * File GETCUST3 from EXTIOSRC in RPGISCOOL

 FCustomerB IF   E           K Disk    Prefix(B_)        1
 FCustomerM IF   E           K Disk    Prefix(M_)

  * Copy in the prototype for the GetCust subprocedure
  /Copy RPGisCool/ExtIOSrc,GetCustPr

  * Constants used in the program
 D MailAddrErr     C                   'Error - Mail Address Missing'
 D SameAsBus       C                   'For Mail Use Business Address'
 D Business        C                   'B'
 D Mailing         C                   'M'
 D Separate        C                   'M'
 D Found           C                   *On
```

```
D NotFound        C                   *Off

 * Structures used to pass Business & Mailing address data to caller
                                                     3
D BusAddr       E DS                  Export
D                                     ExtName(CustomerB) Prefix(B_)

D MailAddr      E DS                  Export
D                                     ExtName(CustomerM) Prefix(M_)

 * Beginning of subprocedure GetCust
P GetCust         B                   Export
D                 PI              N
D  CustNum                            Like(B_Cust#) 2

 * Search for Business address
C     CustNum       Chain     CustomerB       4

 * If the record is found set up data and retrieve
 * Mailing address if MailFlag indicates that one is available
C                   If        %Found(CustomerB)

C                   If        B_MailFlag <> Separate
C                   Eval      MailAddr = *Blanks
C                   Eval      M_CusName = SameAsBus

C                   Else
 * Retrieve mailing address record
C     CustNum       Chain     CustomerM       5

C                   If        %Found(CustomerM)
C                   Else
 * Error - no mailing record found - set up error text
C                   Eval      MailAddr = *Blanks
C                   Eval      M_CusName = MailAddrErr

C                   EndIf
C                   EndIf
 * We have retrieved at least the Business address so return found
C                   Return    Found

C                   Else
 * Requested Customer not found - set up error text and return
C                   Eval      BusAddr = *Blanks
C                   Eval      MailAddr = *Blanks
C                   Eval      B_CusName = NotFound
C                   Return    NotFound
C                   EndIf

P GetCust         E
```

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. We used the same *Prefix* entry later for the data structures BusAddr and MailAddr so that the data from each file is read directly into its associated structure 3.

2 Next we had to modify the definition of CustNum in the procedure interface parameter list to reference the field B_Cust# since the field Cust# which it referenced in GETCUST2 does not exist in this version.

Note: We were 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 Next, we modified the CHAIN operation to use CustNum as the key since there is no longer a need for a keylist. We also removed the key list and the EVAL

that set up the ADDRFLAG field. 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.

6.1.5.2 Changes to SHOWCUST

The time has come to test our earlier claim that having externalized the I/O will minimize the impact of the changes to the calling programs even though we only have one calling program.

Here's the proof. The only change 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 you can see from the following highlighted items:

```
D BusAddr       E DS                  Import
D                                     ExtName(CustomerB) Prefix(B_)

D MailAddr      E DS                  Import
D                                     ExtName(CustomerM) Prefix(M_)
```

Try it yourself

To recreate this example on your system, you need to compile the subprocedure and the main program using 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 we were to 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 would we have to do to achieve this?

We haven't coded a solution to this particular problem, but have offered it as an exercise for you. However, we will make one suggestion. 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 we have convinced you that externalizing your I/O can significantly reduce the effort required to implement design changes in your database. It can also offer other, perhaps unexpected, benefits.

For example, with our I/O operations spread throughout the code, we would probably never add additional logic to our programs to determine if the record being requested had already been read and locked by the previous request. If all I/O for the file is in one place, such a change is trivial, this has the potential to offer significant performance benefits, particularly in a batch environment.

6.2 Replacing indicators with built-in functions

RPG operations on database files, like CHAIN or READ, have always required the use of resulting indicators to signal specific conditions that could arise on such operations. Indicators were used to signal not found, end of file, or error conditions.

Starting with V4R2 a group of built-in functions is available for testing such conditions. You can completely avoid the use of resulting indicators in the file operations. We recommend that you use these new functions since they provide far more readable code than the cryptic use of numbered indicators. Refer to Table 78 for a summary of all of the new built-ins.

Table 78. Built-in functions for file operations

Function	Parameter	Parameter required?	Result type
%EOF	File name	No	Indicator
%EQUAL	File name	No	Indicator
%ERROR	None	N/A	Indicator
%FOUND	File name	No	Indicator
%OPEN	File name	Yes	Indicator
%STATUS	File name	No	Unsigned integer(5,0)

6.2.1 %EOF(FileName)

%EOF returns *On if the most recent read operation or write to a subfile ended in an end of file or beginning of a file condition. Otherwise, it returns *Off.

The file operations that set %EOF condition are:

- READ, READC, READE, READP, READPE
- WRITE to the subfile

The following code snippet illustrates the use of a %EOF built-in function:

```
C                   Read      MyFile
C                   DoW       Not %EOF(MyFile)
C*  Some program logic
C                   Read      MyFile
C                   EndDo
```

Compare this with the same code written in RPG/400:

```
C                     READ MYFILE                   90
C           *IN90     DOWEQ'0'
C*  Some program logic
C                     READ MYFILE                   90
C                     END
```

6.2.2 %EQUAL(FileName)

%EQUAL returns *On if the most recent relevant operation found an exact match. Otherwise, it returns *Off.

Currently SETLL and LOOKUP are the only operations that set the %EQUAL condition.

The following code snippet illustrates the use of the %EQUAL built-in function:

```
C     SearchKey     SetLL     MyFile
C                   If        %EQUAL(MyFile)
C                   Read      MyFile
```

6.2.3 %FOUND(FileName)

%FOUND returns *On if the most recent relevant file operation found a record, a string operation found a match, or a search operation found an element. Otherwise, this function returns *Off.

The operations that set the %FOUND condition are:

- File operations CHAIN, DELETE, SETGT, SETLL
- String operations SCAN, CHECK, CHECKR
- Table operation LOOKUP

The following code snippet illustrates the use of the %FOUND built-in function:

```
C     SearchKey     Chain     MyFile
C                   If        %FOUND(MyFile)
C* Some program logic
C                   EndIf
```

6.2.4 %OPEN(FileName)

%OPEN returns *On if the specified file is open. A file is considered "open" if it has been opened by the RPG program during initialization or by an OPEN operation, and has not subsequently been closed. If the file is conditioned by an external indicator and the external indicator was off at program initialization, the file is considered closed, and %OPEN returns *Off.

The following code snippet illustrates the use of the %OPEN built-in function:

```
FMyFile    IF   E             Disk    UsrOpn
C* If the file is not open, do it now
C                   If        Not %OPEN(MyFile)
C                   Open      MyFile
C                   EndIf
```

6.2.5 %ERROR

%ERROR returns *On if the most recent operation, specified with the extender "E", resulted in an error condition. This is the same as the error indicator being set on for the operation. Before an operation with the extender "E" specified begins, %ERROR is set to return *Off and remains unchanged following the operation if no error occurs. All operations that allow an error indicator can also set the %ERROR built-in function. The CALLP operation can also set %ERROR.

Do not forget to specify error handling extender (E) on the file operations if you want to use the %ERROR built-in function.

6.2.6 %STATUS(FileName)

%STATUS returns the most recent value set for the program or file status. It is set whenever the program status or any file status changes, usually when an error occurs.

If %STATUS is used without the optional file name parameter, it returns the program or file status most recently changed. If a file is specified, the value contained in the *STATUS field of the INFDS data structure for the specified file is returned. The INFDS data structure does not have to be specified for the file.

%STATUS starts with a return value of 00000 and is reset to 00000 before any operation with an (E) extender specified begins.

The following code snippet illustrates the use of the %ERROR and %STATUS built-in functions:

```
FMyFile    IF   E             Disk    UsrOpn
D ErrMsg1         S             20    Inz('Error - File opened')
D ErrMsg2         S             20    Inz('Error - File locked')
D ErrMsg3         S             20    Inz('Error - File closed')
D ErrMsg4         S             20    Inz('End of file reached')
D ErrMsg5         S             20    Inz('Unexpected error')
 * Open the file and check for status code
C                   Open(E)   MyFile
C                   Select
C                   When      %Status(MyFile) = 1215
C     ErrMsg1       Dsply
C                   When      %Status(MyFile) = 1217
C     ErrMsg2       Dsply
C                   EndSl
 * Read the file and check for status code
C                   DoU       %Eof(MyFile)
C                   Read(E)   MyFileR
C                   Select
C                   When      %Error
C                   If        %Status(MyFile) = 1211
C     ErrMsg3       Dsply
C                   Else
C     ErrMsg5       Dsply
C                   EndIf
 * Check for the end of file
C                   When      %Eof(MyFile)
C     Errmsg4       Dsply
C                   Other
 * Some program logic
C                   EndSl
C                   EndDo
```

6.2.7 Indicator data structure

The INDDS keyword lets you associate a data structure name with the INDARA indicators for a workstation or printer file. This data structure contains the conditioning and response indicators passed to and from data management for the file, and is called an indicator data structure.

When using an indicator data structure, you must follow these rules:

- This keyword is allowed only for externally described PRINTER files and externally and program-described WORKSTN files.
- File must be defined using the DDS keyword INDARA.
- The data structure name must be defined as a data structure on the D specifications and can be a multiple-occurrence data structure.
- The length of the indicator data structure is always 99.

- The indicator data structure is initialized by default to all zeros (0).
- The same data structure name may be associated with more than one file.

The following code snippet illustrates the use of the indicator data structure in conjunction with file indicators as well as program internal indicators:

```
FDispFile  CF   E             WorkStn IndDS(DispInd)
 *
D DispInd         DS
 * Response indicators
D  Exit                  3      3N
D  Cancel               12     12N
 * Conditioning indicators
D  DateError            30     30N
D  StrDatError          31     31N
D  EndDatError          32     32N
 * Date validity checking
C                   Eval      StrDatError = StartDate < Today
C                   Eval      EndDatError = EndDate < StartDate
C                   Eval      DateError = StrDatError Or EndDatError
C                   Exfmt     MyScreen
 * Exit the program
C                   If        Exit Or Cancel
```

6.3 Embedded SQL

Instead of using AS/400 native database file operations, like READ, CHAIN, UPDATE, and DELETE, we can embed SQL statements in our RPG IV program and use them to process records in AS/400 database files. SQL is the industry standard for database access and control, and is used by customers on different platforms. The reasons for using embedded SQL in RPG IV programs on the AS/400 system could be:

- Customers 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 6.3.4, “Using a cursor” on page 333.
- SQL operations are performed by a query optimizer, which is enhanced with each new release, and automatically takes advantages of new database technologies.
- Applications migration from or to the AS/400 system is easier if the applications are written using standard language like SQL.

Source code that contains embedded SQL statement must be first 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 AS/400 (5769-ST1), which must be available during the application development. The runtime support is included in the operating system.

The request for additional chargeable software could be a reason for not using an embedded SQL. In that case, you can try to use Call Level Interface APIs, described later in 6.5, “Call Level Interface” on page 348. These system APIs allow the use of SQL statements in RPG IV program, without needing a SQL preprocessor.

6.3.1 Rules for embedding SQL statements

Use the following rules when writing an RPG IV program with embedded SQL statements:

- Enter your SQL statements on the C specification.
- Start your SQL statements using the delimiter /EXEC SQL in positions 7 through 15, with the "/" (slash) in position 7.
- You can start entering your SQL statements on the same line as the starting delimiter or on the new line.
- Use the continuation line delimiter, a "+" (plus sign) in position 7, to continue your statements on any subsequent lines.
- Use the ending delimiter /END-EXEC in positions 7 through 15, with the "/" (slash) in position 7, to signal the end of your SQL statements.

Here is an example of an embedded SQL UPDATE statement:

```
C/Exec Sql
C+     Update Parts
C+            Set PartDes = :DspDes,
C+                PartQty = :DspQty,
C+                PartPrc = :DspPrc,
C+                PartDat = :DspDat
C+            Where PartNum = :DspNum
C/End-Exec
```

The source member containing the RPG IV program with embedded SQL statements must be of type SQLRPGLE. This denotes to PDM options 14 or 15 to execute the CL command CRTSQLRPGI, which is required to call the SQL preprocessor.

6.3.2 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 have 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 PDM option 14 (Compile) or 15 (Create module).

6.3.3 Error and exception handling

SQL does not communicate directly with the end user, but rather 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

6.3.3.1 SQL Communication Area

The SQL preprocessor automatically includes the SQLCA (SQL Communication Area) in the D specifications of the RPG IV program prior to the first C specification. Therefore, it is not necessary to code INCLUDE SQLCA in the source program.

The SQLCA, included in ILE RPG program, contains the following fields:

```
D*      SQL Communications area
D SQLCA            DS
D  SQLAID                 1      8A   INZ(X'0000000000000000')
D  SQLABC                 9     12B 0
D  SQLCOD                13     16B 0
D  SQLERL                17     18B 0
D  SQLERM                19     88A
D  SQLERP                89     96A
D  SQLERRD               97    120B 0 DIM(6)
D  SQLERR                97    120A
D   SQLER1               97    100B 0
D   SQLER2              101    104B 0
D   SQLER3              105    108B 0
D   SQLER4              109    112B 0
D   SQLER5              113    116B 0
D   SQLER6              117    120B 0
D  SQLWRN               121    131A
D   SQLWN0              121    121A
D   SQLWN1              122    122A
D   SQLWN2              123    123A
D   SQLWN3              124    124A
D   SQLWN4              125    125A
D   SQLWN5              126    126A
D   SQLWN6              127    127A
D   SQLWN7              128    128A
D   SQLWN8              129    129A
D   SQLWN9              130    130A
D   SQLWNA              131    131A
D  SQLSTT               132    136A
D*  End of SQLCA
```

The SQLCOD and SQLSTT values are set by the database manager after each SQL statement is executed. 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 your 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 a lot of other fields with specific information relating to the executed SQL statement.

6.3.3.2 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 SQLSTT and SQLCOD 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 looks like this:

```
C/Exec Sql
C+     WhenEver Condition Action
C/End-Exec
```

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.

6.3.4 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 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 sometimes 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 SQL result table to a subfile so the end user can see them. To access a result table, SQL provides technique called *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.

Statements related to using a cursor include:

- DECLARE CURSOR statement defines the name of the cursor and specifies the rows to be retrieved with the embedded SELECT statement.
- OPEN statement opens the cursor for use within the program. The cursor must be opened before any rows can be retrieved.
- FETCH statement retrieves rows from the cursor's result table or positions the cursor on another row.
- CLOSE statement closes the cursor.

The following code snippets illustrates the use of cursor:

```
C/Exec Sql
C+     Declare C1 Cursor For
C+        Select * From Parts
```

```
C+                 Order by PartNum
C+                 For Fetch Only
C/End-Exec

C/Exec Sql
C+     Open C1
C/End-Exec

C/Exec Sql
C+     Fetch C1 Into :SflStr
C/End-Exec

C/Exec Sql
C+     Close C1
C/End-Exec
```

SQL supports two types of cursors: *serial* and *scrollable*. The type of cursor determines the positioning methods that can be used with the cursor.

6.3.4.1 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 your 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, close the cursor. You cannot access any rows in the result table after you reach the end-of-data. To use the cursor again, you must first close the cursor and then re-issue the OPEN statement.

6.3.4.2 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 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. Default if no position 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 n	Positions the cursor for n rows relative to the current position.

6.3.5 An embedded SQL program example

To illustrate the coding of embedded SQL statements in RPG IV program, we use a simple example of file maintenance program, where we 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, we provide you the DDS definitions for both the database and display file.

The database file PARTS contains five fields, one of them is used as a key:

```
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 contains two records and a subfile:

```
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                   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 us 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 Parts file and is populated using the FETCH statement.

Based on the function key that is pressed, different SQL statements in the program are executed.

6.3.6 Source code for SQLEMBED program

Now we can analyze our RPG IV program with embedded SQL statements:

```
 ***********************************************************************
 *  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 (PDM Option=14)
 *  or use command CRTSQLRPGI with COMMIT(*NONE)
 ***********************************************************************
 * Display file with subfile
FDspFil1   CF   E             WorkStn IndDS(DispInd)                  1
F                                     SFile(SflRec1:RecNum)
 *---------------------------------------------------------------------
D RecNum          S              3  0
 *
 * Indicator data structure for display file indicators
 *
D DispInd         DS                                                2
D  Exit                  3      3N
D  ListAllRec            4      4N
D  InsertRec             5      5N
D  UpdateRec             6      6N
D  DeleteRec             7      7N
D  Cancel               12     12N
D  InvalidRec           55     55N
D  ClearSfl             66     66N
 *
 * Host structure to simplify host variables in SQL statement
 *
D HostStr         DS                                                3
D  DspNum                        5  0
D  DspDes                       25
D  DspQty                        5  0
D  DspPrc                        6  2
D  DspDat                         D
 *---------------------------------------------------------------------
C                   Exfmt     DspRec1
 * Loop begin
C                   DoW       Not (Exit Or Cancel)
C                   Eval      InvalidRec = *Off
C                   Select
 *
 * Insert new record into file Parts
 *
C                   When      InsertRec
C                   Clear                   HostStr
C                   Eval      DspNum = PartNo
C                   Exfmt     DspRec2
 *
C/Exec Sql
C+     Insert Into Parts                                            4
C+            Values (:HostStr)
C/End-Exec
 *
C                   If        SqlStt = '23505'
C                   Eval      InvalidRec = *On
C                   Endif
 *
 * List all records from file Parts using subfile
 *
C                   When      ListAllRec
C                   Eval      RecNum = 0
C                   Eval      ClearSfl = *On
C                   Write     SflRec2
C                   Eval      ClearSfl = *Off
 *
C/Exec Sql
C+     Declare C1 Cursor For                                        8
C+        Select * From Parts
```

```
C+                 Order by PartNum
C+                 For Fetch Only
C/End-Exec
 *
C/Exec Sql
C+     Open C1                                                 9
C/End-Exec
 *
C/Exec Sql
C+     Fetch C1 Into :HostStr                                  10
C/End-Exec
 *
C                   DoW       SqlStt <> '02000'
C                   Eval      RecNum = RecNum + 1
C                   Write     SflRec1
 *
C/Exec Sql
C+     Fetch C1 Into :HostStr                                  10
C/End-Exec
 *
C                   EndDo
 *
C/Exec Sql
C+     Close C1                                                11
C/End-Exec
 *
C                   Exfmt     SflRec2
 *
 * Display selected record from file Parts
 *
C                   Other
 *
C/Exec Sql
C+     Select * Into :HostStr                                  7
C+              From Parts
C+              Where PartNum = :PartNo
C/End-Exec
 *
C                   If        SqlStt = '02000'
C                   Eval      InvalidRec = *On
C                   Else
C                   Exfmt     DspRec2
C                   Select
C                   When      Exit Or Cancel
C                   Leave
 *
 * Update selected record in file Parts
 *
C                   When      UpdateRec
 *
C/Exec Sql
C+     Update Parts                                            5
C+            Set PartDes = :DspDes,
C+                PartQty = :DspQty,
C+                PartPrc = :DspPrc,
C+                PartDat = :DspDat
C+            Where PartNum = :DspNum
C/End-Exec
 *
 * Delet selected record from file Parts
 *
C                   When      DeleteRec
 *
C/Exec Sql
C+     Delete From Parts                                       6
C+            Where PartNum = :DspNum
C/End-Exec
 *
C                   EndSl
C                   EndIf
C                   EndSl
C                   Exfmt     DspRec1
C                   EndDo
 * Loop end
C                   Eval      *INLR = *On
```

SQLEMBED program notes

1 On the F 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 D specifications, we use the indicator data structure DISPIND, where all display file response and conditioning indicators are defined with meaningful names. This improves the readability of our program.

3 Program variables used in SQL statements are called host variables and must be preceded by a ":" (colon). To simplify the writing of SQL statements, we recommend that you define the data structure on D specifications, which contains all host variables needed by the SQL statement. Such data structure is known as a *host structure* and can be also externally defined.

4 For adding a new record into the file, the program uses the INSERT statement with field values taken from the host structure. After inserting, we have to check SQLSTT for 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. The record was previously read, and there is no need to check the return code.

6 A record delete is performed with statement DELETE. The record was previously read, and there is no need to check the return code.

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 record is identified by a WHERE clause. By checking that SQLSTT equals the value '02000', we can identify that the no record found an exception.

8 To read all records from the file and put them into subfile, we have to use the cursor technique. The cursor should be first declared and related to the corresponding SELECT statement.

9 The OPEN statement actually 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, we didn't specify any validation of the %EOF (end of file) condition for the subfile, which could 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

Create the display file from the source:

```
CRTDSPF FILE(RPGISCOOL/DSPFIL1) SRCFILE(RPGISCOOL/DBSRC) SRCMBR(DSPFIL1)
```

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
```

Create the program and the program call:

```
CRTSQLRPGI OBJ(RPGISCOOL/SQLEMBED) SRCFILE(RPGISCOOL/DBSRC) COMMIT(*NONE)
CALL PGM(SQLEMBED)
```

6.4 Stored procedures

Stored procedure support is a function of DB2 SQL for AS/400. It provides a way for an SQL application to define and then invoke a procedure through SQL statements. Stored procedures can be used in both distributed (client/server) and non-distributed DB2 SQL for AS/400 applications.

One of the big advantages in using stored procedures is that 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 significantly reduce the data transfer between the client and server and consequently improve the performance of the distributed application.

You may 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 invoking the procedure

6.4.1 Creating an external procedure

To create an external procedure, we use SQL statement CREATE PROCEDURE, which defines 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 since 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.

6.4.2 Creating an SQL procedure

To create an SQL procedure, we use the same SQL statement CREATE PROCEDURE, which defines:

- 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

The following example 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:

```
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 executed.

6.4.3 Invoking a stored procedure and returning the completion status

To invoke a stored procedure we 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. Here is an example of how to call a stored procedure and pass two arguments:

```
/Exec Sql
CALL mylib/procname (:PARTNUM, :PARTDES)
/End-Exec
```

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 prior to 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 invokes the procedure.

6.4.4 A stored procedure example

To illustrate the use of stored procedure we modified the example that we used in 6.3.6, “Source code for SQLEMBED program” on page 336. Embedded SQL statements are replaced with SQL CALL statements to call stored procedures. We created 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.

6.4.4.1 Source code for program SPRCRUN

The logic of the program SPRCRUN is the same as of the program SQLEMBED in 6.3.6, “Source code for SQLEMBED program” on page 336. It uses the display file DSPFIL1 to communicate with the end user.

```
 **********************************************************************
 *  Filename 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)
 **********************************************************************
 * Display file with subfile
FDspFil1   CF   E             WorkStn IndDS(DispInd)                  1
F                                     SFile(SflRec1:RecNum)
 *--------------------------------------------------------------------
D RecNum          S              3  0
 * Indicator data structure for display file indicators
 *
D DispInd         DS                                                 2
D  Exit                   3      3N
D  ListAllRec             4      4N
D  InsertRec              5      5N
D  UpdateRec              6      6N
D  DeleteRec              7      7N
D  Cancel                12     12N
D  InvalidRec            55     55N
D  ClearSfl              66     66N
 * Action parameter and its meaning
 *
D Action          S              1                                   3
D  SingleRec      C                   CONST('S')
D  FirstRec       C                   CONST('F')
D  NextRec        C                   CONST('N')
D  EndOfFile      C                   CONST('E')
D  UpdRecord      C                   CONST('U')
D  AddRecord      C                   CONST('I')
D  Error          C                   CONST('X')
 *--------------------------------------------------------------------
 * Stored Procedures SPrcSel written with embedded SQL
 *
C/Exec Sql
C+     Create Procedure RpgIsCool/SPrcSel                               4
C+        (InOut Action Char(1), InOut PartNum Numeric(5 , 0),
C+         Out PartDes Char(25), Out PartQty Numeric(5 , 0),
C+         Out PartPrc Numeric(6 , 2), Out PartDat Date)
C+        Language RPGLE
C+        Modifies SQL Data
C+        External Name RpgIsCool/ProgSel
C/End-Exec * Stored Procedures SPrcUpd written with native file operations
 *
C/Exec Sql
C+     Create Procedure RpgIsCool/SPrcUpd                              5
C+        (InOut Action Char(1), In PartNum Numeric(5 , 0),
C+         In PartDes Char(25), In PartQty Numeric(5 , 0),
C+         In PartPrc Numeric(6 , 2), In PartDat Date)
C+        Language RPGLE
C+        Modifies SQL Data
C+        External Name RpgIsCool/ProgUpd
C/End-Exec
 * Stored Procedures SPrcDel written in SQL
 *
C/Exec Sql
C+     Create Procedure RpgIsCool/SPrcDel                                6
C+        (In PartNo Numeric(5 , 0))
C+        Language SQL
C+        Modifies SQL Data
C+        Delete From Parts Where PartNum = PartNo
C/End-Exec
 *
C                   Exfmt     DspRec1
 * Loop begin
C                   DoW       Not (Exit Or Cancel)
C                   Eval      InvalidRec = *Off
C                   Select
 *
 * Insert new record into file Parts
 *
C                   When      InsertRec
C                   Clear                   DspDes
C                   Clear                   DspQty
C                   Clear                   DspPrc
C                   Clear                   DspDat
C                   Eval      DspNum = PartNo
C                   Exfmt     DspRec2
C                   Eval      Action = AddRecord
 *
C/Exec Sql
C+     Call SPrcUpd                                                    7
C+          (:Action, :PartNo, :DspDes, :DspQty, :DspPrc, :DspDat)
C/End-Exec
 *
C                   If        Action = Error
C                   Eval      InvalidRec = *On
C                   Endif
 *
 * List all records from file Parts using subfile
 *
C                   When      ListAllRec
C                   Eval      RecNum = 0
C                   Eval      ClearSfl = *On
C                   Write     SflRec2
C                   Eval      ClearSfl = *Off
C                   Eval      Action = FirstRec
 *
C/Exec Sql
C+     Call SPrcSel                                                    11
C+          (:Action, :DspNum, :DspDes, :DspQty, :DspPrc, :DspDat)
C/End-Exec
 *
C                   DoW       Action <> EndOfFile
C                   Eval      RecNum = RecNum + 1
C                   Write     SflRec1
C                   Eval      Action = NextRec
 *
C/Exec Sql
C+     Call SPrcSel                                                    12
C+          (:Action, :DspNum, :DspDes, :DspQty, :DspPrc, :DspDat)
C/End-Exec
 *
C                   EndDo
C                   Exfmt     SflRec2
 *
 * Display selected record from file Parts
 *
C                   Other
C                   Eval      Action = SingleRec
C                   Eval      DspNum = PartNo
 *
C/Exec Sql
C+     Call SPrcSel                                                     10
C+          (:Action, :DspNum, :DspDes, :DspQty, :DspPrc, :DspDat)
C/End-Exec
```

```
 *
C                   If        Action = Error
C                   Eval      InvalidRec = *On
C                   Else
C                   Exfmt     DspRec2
C                   Select
C                   When      Exit Or Cancel
C                   Leave
 *
 * Update selected record in file Parts
 *
C                   When      UpdateRec
C                   Eval      Action = UpdRecord
 *
C/Exec Sql
C+     Call SPrcUpd                                                   8
C+          (:Action, :PartNo, :DspDes, :DspQty, :DspPrc, :DspDat)
C/End-Exec
 *
 * Delet selected record from file Parts
 *
C                   When      DeleteRec
 *
C/Exec Sql
C+     Call SPrcDel (:PartNo)                                         9
C/End-Exec
 *
C                   EndSl
C                   EndIf
C                   EndSl
C                   Exfmt     DspRec1
C                   EndDo
 * Loop end
C                   Eval      *INLR = *On
```

SPRCRUN program notes

1 On the F 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 D specifications, we use 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.

4 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 program PROGSEL. As you will see later, this program contains the embedded SQL statements SELECT, FETCH, OPEN and CLOSE cursor to read data from the PARTS file.

5 Another stored procedure, SPRCUPD, is created. It requires one input/output parameter and five input parameters, and relates to the RPG IV program PROGUPD. This program contains only native file operations CHAIN, WRITE, and UPDATE to show you that any program can be declared and used as a stored procedure.

6 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 using interactive SQL (STRSQL). Created stored procedures are cataloged in the SQL catalog, and can be used from any program using the SQL CALL statement.

7 Stored procedure SPRCUPD is called to insert a record.

8 Stored procedure SPRCUPD is called to update a record.

9 Stored procedure SPRCDEL is called to delete a record.

10 Stored procedure SPRCSEL is called to read a single record.

11 Stored procedure SPRCSEL is called to read the first record.

12 Stored procedure SPRCSEL is called to read the next record.

6.4.4.2 Source code for PROGSEL program

This program is called as a stored procedure SPRCSEL. It contains the embedded SQL statements SELECT, FETCH, OPEN, and CLOSE cursor.

```
 ***********************************************************************
 *  Filename PROGSEL from DBSRC in RPGISCOOL
 *  RPG program PROGSEL with embedded SQL, used as stored procedure
 *
 *  Examples of SELECT and FETCH
 *
 *  Compile this source member as program PROGSEL (PDM Option=14)
 *  or use command CRTSQLRPGI with COMMIT(*NONE)
 ***********************************************************************
 * Prototype and entry parameter definition
 *
D ProgSel         PR                  EXTPGM('PROGSEL')                1
D  Action                        1
D  PartNo                        5S 0
D  PartDs                       25
D  PartQy                        5S 0
D  PartPr                        6S 2
D  PartDt                         D
 *
D ProgSel         PI                                                   2
D  Action                        1
D  PartNo                        5S 0
D  PartDs                       25
D  PartQy                        5S 0
D  PartPr                        6S 2
D  PartDt                         D
 * Meaning of Action parameter
 *
D  SingleRec      C                   CONST('S')                       3
D  FirstRec       C                   CONST('F')
D  NextRec        C                   CONST('N')
D  EndOfFile      C                   CONST('E')
D  Error          C                   CONST('X')
 *----------------------------------------------------------------------
 *
C                   Select
C                   When      Action = SingleRec
 *
 * Read single record from file Parts
 *
C/Exec Sql
C+     Select * Into :PartNo, :PartDs, :PartQy, :PartPr, :PartDt      4
C+             From Parts
C+             Where PartNum = :PartNo
C/End-Exec
 *
C                   If        SqlStt = '02000'
C                   Eval      Action = Error
C                   Endif
C                   When      Action = FirstRec
 *
C/Exec Sql
C+     Declare C1 Cursor For                                          5
C+        Select * From Parts
C+                Order by PartNum
C+                For Fetch Only
C/End-Exec
 *
C/Exec Sql
C+     Open C1                                                        6
```

```
C/End-Exec
 *
 * Read first record from file Parts
 *
C/Exec Sql
C+     Fetch C1 Into :PartNo, :PartDs, :PartQy, :PartPr, :PartDt      7
C/End-Exec
 *
C                   When      Action = NextRec
 *
 * Read next record from file Parts
 *
C/Exec Sql
C+     Fetch C1 Into :PartNo, :PartDs, :PartQy, :PartPr, :PartDt      7
C/End-Exec
 *
C                   If        SqlStt = '02000'
C                   Eval      Action = EndOfFile
 *
C/Exec Sql
C+     Close C1                                                       8
C/End-Exec
 *
C                   Endif
C                   EndSl
C                   Return
```

PROGSEL program notes

1 The prototype and procedure interface for the main procedure replace the *ENTRY parameter list required in a called program.

2 The procedure interface defines the required parameters for this program.

3 To define meaningful names for different values of ACTION parameter, we recommend that you use constants.

4 To read a single record, we use the SELECT statement with the INTO clause. If the record doesn't 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.

6.4.4.3 Source code for the PROGUPD program

This program is called as a stored procedure SPRCUPD. It contains only the native file operations CHAIN, WRITE, and UPDATE and illustrates that any program can be declared and used as a stored procedure.

```
*********************************************************************
 *  Filename PROGUPD from DBSRC in RPGISCOOL
 *  RPG program PROGUPD w/o embedded SQL, used as stored procedure
 *
 *  Examples of CHAIN, WRITE and UPDATE
 *
 *  Compile this source member as program PROGUPD (PDM Option=14)
 *  or use command CRTBNDRPG
 *********************************************************************
FParts     UF A E           K Disk
 *-------------------------------------------------------------------
 * Prototype and entry parameter definition
 *
D ProgUpd         PR                  EXTPGM('PROGUPD')              1
D  Action                       1
D  PartNo                       5S 0
D  PartDs                      25
D  PartQy                       5S 0
D  PartPr                       6S 2
D  PartDt                        D
 *
D ProgUpd         PI                                                  2
D  Action                       1
D  PartNo                       5S 0
D  PartDs                      25
D  PartQy                       5S 0
D  PartPr                       6S 2
D  PartDt                        D
 * Meaning of Action parameter
 *
D  UpdRecord      C                   CONST('U')                     3
D  AddRecord      C                   CONST('I')
D  Error          C                   CONST('X')
 *-------------------------------------------------------------------
 * Read a record from file Parts
 *
C     PartNo        Chain     Parts                                  4
C                   Eval      PartNum = PartNo
C                   Eval      PartDes = PartDs
C                   Eval      PartQty = PartQy
C                   Eval      PartPrc = PartPr
C                   Eval      PartDat = PartDt
C                   Select
C                   When      Action = UpdRecord And %FOUND(Parts)
 * Update a record in file Parts
C                   Update    PartR
C                   When      Action = AddRecord And Not %FOUND(Parts)
 * Write a record into file Parts
C                   Write     PartR
C                   Other
 * Return error in other cases
C                   Eval      Action = Error
C                   EndSl
C                   Return
```

PROGUPD program notes

1 Prototype and procedure interface for the main procedure replace the *ENTRY parameter list, which is required in a called program.

2 The procedure interface defines the required parameters for this program.

3 To define meaningful names for different values of the ACTION parameter, we recommend that you use constants.

4 With the CHAIN operation, we first check if the record exists, and then update the record, write the record, or return an error through the ACTION parameter.

Try it yourself

You can try this example by compiling the code from this section on your AS/400 system. Use the following commands to create the programs:

```
CRTSQLRPGI OBJ(RPGISCOOL/SPRCRUN) SRCFILE(RPGISCOOL/DBSRC) +
COMMIT(*NONE) DATFMT(*ISO)
CRTSQLRPGI OBJ(RPGISCOOL/PROGSEL) SRCFILE(RPGISCOOL/DBSRC)
CRTBNDRPG PGM(RPGISCOOL/PROGUPD) SRCFILE(RPGISCOOL/DBSRC)
```

To run the program, enter the following command:

```
CALL PGM(SPRCRUN)
```

6.5 Call Level Interface

DB2 Call Level Interface (CLI) is a callable SQL programming interface that is supported in all DB2 environments except for DB2 for MVS and DB2 for VSE and VM. 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 AS/400 system, this interface is available to any of the ILE languages, including RPG IV.

DB2 CLI also provides full Level 1 Microsoft's 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 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.

6.5.1 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 executed. In contrast, a DB2 CLI application does not require precompilation or binding, but instead uses a standard set of functions to execute SQL statements and related services at runtime.

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 runtime.

DB2 CLI can execute any SQL statement that can be prepared dynamically in embedded SQL. This is guaranteed because DB2 CLI doesn't actually execute the SQL statement itself, but passes it to the DBMS for dynamic execution.

6.5.1.1 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 executing 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, since 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.
- As important as any other technical issue, the DB2 CLI support is included in the base OS/400 support and is available as a part of system APIs.

6.5.1.2 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 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 using DB2 CLI, but rather that the application will no longer be as portable.

Another important consideration is the performance comparison between a dynamic and static SQL. A dynamic SQL is prepared at runtime, while a static SQL is prepared at the precompile stage. Since preparing statements requires additional processing time, static SQL may 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 may make one alternative seem more intuitive than the other.

6.5.2 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, while others may 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 shown in Figure 39 on page 350.

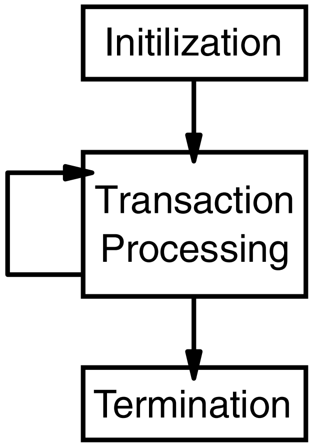


Figure 39. Conceptual view of a DB2 CLI application

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 identified by unique handles. After the resources have been 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.

6.5.3 Initialization and termination

Figure 40 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 41 on page 352.

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 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, 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 associated with a connection managed by DB2 CLI. This 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 managed by DB2 CLI. This 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 SQLAllocStmt() and freed by calling SQLFreeStmt(), and must be associated with a connection handle.

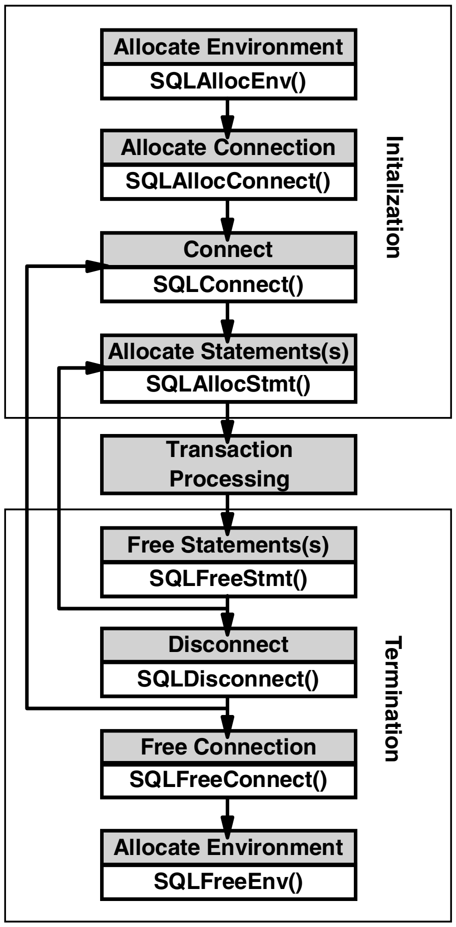


Figure 40. Conceptual view of initialization and termination tasks

6.5.4 Transaction processing

Figure 41 on page 352 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 execute SQL statements.
3. Process the results.
4. Free the statement handles.
5. Commit or rollback.

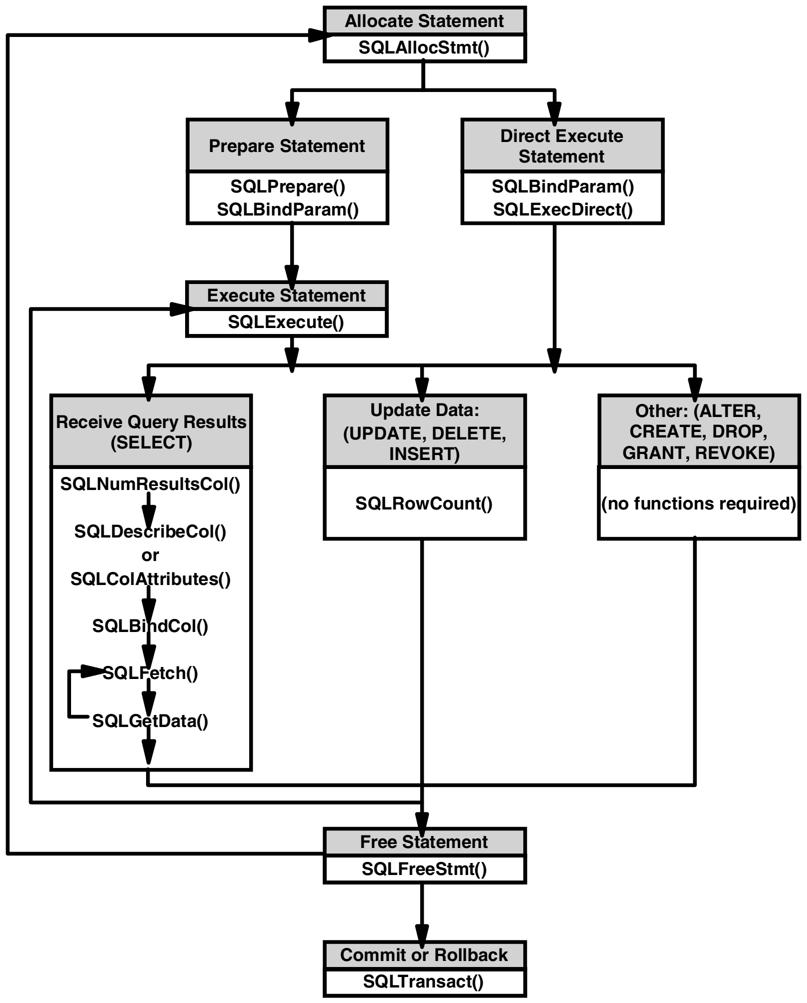


Figure 41. Transaction processing

6.5.4.1 Allocating statement handles

To run a statement, allocate a statement handle using SQLAllocStmt(). Each statement handle must be associated with a connection handle.

6.5.4.2 Preparation and execution of SQL statements

Once a statement handle has been allocated, there are two methods of specifying and executing SQL statements:

- Prepare then execute in order:
  1. Call SQLPrepare() with an SQL statement as an argument
  2. Call SQLBindParam(), if the SQL statement contains parameter markers
  3. Call SQLExecute()

- Execute direct:
  1. Call SQLBindParam() if the SQL statement contains parameter markers
  2. Call SQLExecDirect() with an SQL statement as an argument

The first method splits the preparation of the statement from the execution. This method is used when:

- The statement is executed repeatedly (usually with different parameter values). This avoids having to prepare the same statement more than once.
- The application requires information about the columns in the result set, prior to statement execution.

The second method combines the preparation step and the execution step into one. This method is used when:

- The statement is executed once. This avoids having to call two functions to execute the statement.
- The application does not require information about the columns in the result set before the statement is executed.

Both execution methods allow the use of parameter markers in place of an expression (or 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 to be substituted when the statement is executed. 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 since only the pointer is passed when SQLBindParam() is called. No data is read from the variable until the statement is executed. This applies to both buffer arguments and arguments that indicate the length of the data in the buffer. Deferred arguments allow 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 AS/400 SQL Call Level Interface,* SC41-5806, for V4R3 and V4R4, you can find the statement that the SQLBindParam() function allows to bind a variable of a different type from that required by the SQL statement. In this case, DB2 CLI should convert the contents of the bound variable to the correct type. For example, the SQL statement may require an integer value, but your application has a string representation of an integer, which should be converted to an integer when the statement is executed. This conversion is not yet possible and should be available in future releases.

6.5.4.3 Processing results

The next step after the statement has been executed 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 isn't necessary. This is because the application knows the structure of the result set and the data types of each column. If the SQL statement was generated at runtime (entered by a user), the application needs to query:

- 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 SQLColAttributes()) after preparing the statement or after executing 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 using SQLBindParam(), 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() can also 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 have been 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 also 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.

6.5.4.4 Freeing statement handles
To end processing for a particular statement handle, you should 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.

6.5.4.5 Commit or rollback
The last step is to either commit or rollback the transaction, using SQLTransact(). A transaction is a recoverable unit of work or a group of SQL statements that can be treated as one atomic operation. This means that all the operations within the group are to be completed (committed) or undone (rolled back), as if they were a single operation.

When using DB2 CLI, transactions are started implicitly with the first access to the database using SQLPrepare(), SQLExecDirect(), or SQLGetTypeInfo(). The transaction ends when you use SQLTransact() to either rollback or commit the transaction. This means that any SQL statements executed between these are treated as one unit of work.

6.5.5 Diagnostic

Diagnostics refers to dealing with warning or error conditions generated within an application. There are two levels of diagnostics 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 79 on page 356 describes all DB2 CLI function return codes.

Table 79. 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 SQLError() 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 SQLError() 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 a class of:

- **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 SQLError() to determine if diagnostic information is available.
- Use SQLSTATEs rather than the native error code.
- To increase your application's portability, only build dependencies 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 will include the IBM defined SQLSTATE). It is also useful for the application to print out the name of the function that returned the error.

6.5.6 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 is unavoidable since DBMS uses SQL data types and the application must use RPG IV data types. This means 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, DB2 CLI needs to know both the source and target data type. This requires the application to identify both data types 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 following code snippet illustrates how symbolic names can be defined:

```
 *******************************************************************
 * Standard SQL data types
 *******************************************************************
DSQL_CHAR         C                   CONST(1)
DSQL_NUMER        C                   CONST(2)
DSQL_DECIM        C                   CONST(3)
DSQL_INTEG        C                   CONST(4)
DSQL_SMINT        C                   CONST(5)
DSQL_FLOAT        C                   CONST(6)
DSQL_REAL         C                   CONST(7)
DSQL_DOUBLE       C                   CONST(8)
DSQL_DATTIM       C                   CONST(9)
DSQL_VARCH        C                   CONST(12)
DSQL_GRAPH        C                   CONST(95)
DSQL_VARGR        C                   CONST(96)
DSQL_DATE         C                   CONST(91)
DSQL_TIME         C                   CONST(92)
DSQL_TIMEST       C                   CONST(93)
DSQL_CD_DAT       C                   CONST(1)
DSQL_CD_TIM       C                   CONST(2)
DSQL_CD_TST       C                   CONST(3)
DSQL_ALLTYP       C                   CONST(0)
```

6.5.7 Functions

All DB2 CLI functions are available as procedures in the service program QSQCLI in the library QSYS. In version V4R4, there are 77 functions. They are described in the IBM manual *DB2 for AS/400 SQL Call Level Interface (ODBC)*, SC41-5806.

We have included here only the descriptions of those functions that we need in our example.

6.5.7.1 SQLAllocEnv(): Allocate environment handle

SQLAllocEnv() allocates an environment handle and associated resources. An application must call this function prior to SQLAllocConnect() or any other DB2 CLI functions. The henv value is passed in all later function calls that require an environment handle as input.

The C syntax is:

```
SQLRETURN SQLAllocEnv (SQLHENV    *phenv);
```

Table 80 shows the parameters for this function.

Table 80. Parameters for the SQLAllocEnv function

Argument	Description	Use	RPG data type	C data type
phenv	Pointer to 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.

6.5.7.2 SQLAllocConnect(): Allocate connection handle

SQLAllocConnect() allocates a connection handle and associated resources within the environment identified by the input environment handle.

SQLAllocEnv() must be called before calling this function.

The C syntax is:

```
SQLRETURN SQLAllocConnect (SQLHENV    henv,
                           SQLHDBC    *phdbc);
```

Table 81 shows the parameters for this function.

Table 81. Parameters for the SQLAllocConnect function

Argument	Description	Use	RPG data type	C data type
henv	Environment handle	Input	Integer(10)	SQLHENV
phdbc	Pointer to 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.

6.5.7.3 SQLConnect(): Connect 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().

The C syntax is:

```
SQLRETURN SQLConnect (SQLHDBC         hdbc,
                      SQLCHAR         *szDSN,
                      SQLSMALLINT     cbDSN,
                      SQLCHAR         *szUID,
                      SQLSMALLINT     cbUID,
                      SQLCHAR         *szAuthStr,
                      SQLSMALLINT     cbAuthStr);
```

Table 82 shows the parameters for this function.

Table 82. 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 database name	Input	Integer(5)	SQLSMALLINT
*szUID	User Id	Input	Pointer	SQLCHAR *
cbUID	Length of User Id	Input	Integer(5)	SQLSMALLINT
*szAuthStr	User password	Input	Pointer	SQLCHAR *
cbAuthStr	Length of password	Input	Integer(5)	SQLSMALLINT

You can define various connection characteristics (options) in the application using SQLSetConnectOption().

The database name must already be defined on the system for the connect to work. On AS/400 system, you can use the Work with Relational Database Directory Entries (WRKRDBDIRE) command to determine which data sources have been defined already and to optionally define additional data sources.

The return codes are SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, and SQL_INVALID_HANDLE.

6.5.7.4 SQLSetConnectOption(): Set connection option

SQLSetConnectOption() sets connection attributes for a particular connection.

The C syntax is:

```
SQLRETURN SQLSetConnectOption (HDBC        hdbc,
                               SQLSMALLINT fOption,
                               SQLPOINTER  vParam);
```

Table 83 shows the parameters for this function.

Table 83. 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 associated with option	Input	Pointer	SQLPOINTER

The SQLSetConnectOption() provides the same function as SQLSetConnectAttr(). Both functions are supported for compatibility reasons.

All connection and statement options set through the 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.

6.5.7.5 SQLAllocStmt(): Allocate a statement handle

SQLAllocStmt() allocates a new statement handle and associates it with the connection 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.

The C syntax is:

```
SQLRETURN SQLAllocStmt (SQLHDBC    hdbc,
                        SQLHSTMT   *phstmt);
```

Table 84 shows the parameters for this function.

Table 84. 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.

6.5.7.6 SQLPrepare(): Prepare 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 has been used with a SELECT statement, SQLFreeStmt() must be called to close the cursor before calling SQLPrepare().

The C syntax is:

```
SQLRETURN SQLPrepare (SQLHSTMT       hstmt,
                      SQLCHAR        *szSqlStr,
                      SQLINTEGER     cbSqlStr);
```

Table 85 shows the parameters for this function.

Table 85. 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 executed 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 to be 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.

6.5.7.7 SQLBindCol(): Bind 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 required. It is called once for each column in the result set that the application needs to retrieve.

SQLPrepare() or SQLExecDirect() is usually called before this function. It may also be necessary to call SQLDescribeCol() or SQLColAttributes().

SQLBindCol() must be called before SQLFetch() to transfer data to the storage buffers specified by this call.

The C syntax is:

```
SQLRETURN SQLBindCol (SQLHSTMT      hstmt,
                      SQLSMALLINT   icol,
                      SQLSMALLINT   fCType,
                      SQLPOINTER    rgbValue,
                      SQLINTEGER    cbValueMax,
                      SQLINTEGER   *pcbValue);
```

Table 86 shows the parameters for this function.

Table 86. 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 variable where to store column data	Output (defer)	Pointer	SQLPOINTER
cbValueMax	Available size of variable	Input	Integer(10)	SQLINTEGER
*pbcValue	Size of data returned into 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 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: `(precision * 256) + 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.

6.5.7.8 SQLBindParam(): Binds a buffer to a parameter marker

SQLBindParam() 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().

The C syntax is:

```
SQLRETURN SQLBindParam (SQLHSTMT    hstmt,
                        SQLSMALLINT  ipar,
                        SQLSMALLINT  fCType,
                        SQLSMALLINT  fSqlType,
                        SQLINTEGER   cbParamDef,
                        SQLSMALLINT  ibScale,
                        SQLPOINTER   rgbValue,
                        SQLINTEGER  *pcbValue);
```

Table 87 shows the parameters for this function.

Table 87. Parameters for the SQLBindParam 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
fCType	Application data type	Input	Integer(5)	SQLSMALLINT
fSqlType	SQL data type	Input	Integer(5)	SQLSMALLINT
cbParamDef	Length of parameter	Input	Integer(10)	SQLINTEGER
ibScale	Number of decimal positions	Input	Integer(5)	SQLSMALLINT
rgbValue	Buffer with actual data for parameter	Input or Output	Pointer	SQLPOINTER
*pbcValue	Value interpreted during execution	Input	Integer(10)	SQLINTEGER *

The return codes are SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, and SQL_INVALID_HANDLE.

6.5.7.9 SQLExecute(): Execute a statement

SQLExecute() executes a statement that was successfully prepared by using SQLPrepare() once or multiple times. The statement is executed using the current values of any application variables that were bound to parameter markers by SQLBindParam().

The C syntax is:

```
SQLRETURN SQLExecute (SQLHSTMT     hstmt);
```

Table 88 shows the parameters for this function.

Table 88. 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 to be 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.

All parameters must be bound before calling SQLExecute(). Once the application has processed the results from the SQLExecute() call, it can execute 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.

6.5.7.10 SQLExecDirect(): Execute a statement directly

SQLExecDirect directly executes the specified SQL statement. The statement can only be executed once. Also, the connected database server must be able to prepare the statement.

The C syntax is:

```
SQLRETURN SQLExecDirect (SQLHSTMT      hstmt,
                         SQLCHAR       *szSqlStr,
                         SQLINTEGER    cbSqlStr);
```

Table 89 shows the parameters for this function.

Table 89. 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 *
cbSqlStr	Length of 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 to be substituted when SQLExecDirect() is called. SQLBindParam() binds (or associates) an application variable to each parameter marker to indicate if 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.

6.5.7.11 SQLFetch(): Fetch 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.

The C syntax is:

```
SQLRETURN SQLFetch (SQLHSTMT  hstmt);
```

Table 90 shows the parameters for this function.

Table 90. Parameters for the SQLFetch function

Argument	Description	Use	RPG data type	C data type
hstmt	Statement handle	Input	Integer(10)	SQLHSTMT

SQLFetch() can only be called if the most recently executed 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() will fail.

If SQLBindCol() has not been 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 have been 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.

6.5.7.12 SQLTransact(): Transaction management

SQLTransact() commits or rolls back the current transaction in the connection. All changes to the database 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.

The C syntax is:

```
SQLRETURN SQLTransact (SQLHENV        henv,
                       SQLHDBC        hdbc,
                       SQLSMALLINT    fType);
```

Table 91 shows the parameters for this function.

Table 91. 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 execute them as part of a new transaction. This 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.

6.5.7.13 SQLError(): Retrieve error information

SQLError() returns the diagnostic information 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.

The C syntax is:

```
SQLRETURN SQLError (SQLHENV       henv,
                    SQLHDBC       hdbc,
                    SQLHSTMT      hstmt,
                    SQLCHAR       *szSqlState,
                    SQLINTEGER    *pfNativeError,
                    SQLCHAR       *szErrorMsg,
                    SQLSMALLINT   cbErrorMsgMax,
                    SQLSMALLINT   *pcbErrorMsg);
```

Table 92 shows the parameters for this function.

Table 92. 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
hstmt	Statement handle	Input	Integer(10)	SQLHSTMT
*szSqlState	SQLSTATE as a string	Output	Pointer	SQLCHAR *

Argument	Description	Use	RPG data type	C data type
*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 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.

6.5.7.14 SQLFreeStmt(): Free (or reset) a statement handle

SQLFreeStmt() ends processing on the statement referenced by the statement handle. Use this function to:

- Close a cursor
- Reset parameters
- Unbind columns from variables
- Drop the statement handle and free the DB2 CLI resources associated with the statement handle

SQLFreeStmt() is called after executing an SQL statement and processing the results.

The C syntax is:

```
SQLRETURN SQLFreeStmt (SQLHSTMT       hstmt,
                       SQLSMALLINT    fOption);
```

Table 93 shows the parameters for this function.

Table 93. 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) 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 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 bound by previous SQLBindCol() calls on this statement handle are released.

- **SQL_RESET_PARAMS**

  All the parameters 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.

6.5.7.15 SQLDisconnect(): Disconnect from a data source

SQLDisconnect() closes the connection associated with the database connection handle. After calling this function, either call SQLConnect() to connect to another database, or call SQLFreeConnect().

The C syntax is:

```
SQLRETURN SQLDisconnect (SQLHDBC    hdbc);
```

Table 94 shows the parameters for this function.

Table 94. 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 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.

6.5.7.16 SQLFreeConnect(): Free connection handle

SQLFreeConnect() invalidates and frees the connection handle. All DB2 CLI resources 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.

The C syntax is:

```
SQLRETURN SQLFreeConnect (SQLHDBC    hdbc);
```

Table 95 shows the parameters for this function.

Table 95. 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.

6.5.7.17 SQLFreeEnv(): Free environment handle

SQLFreeEnv() invalidates and frees the environment handle. All DB2 CLI resources 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.

C syntax:

```
SQLRETURN SQLFreeEnv (SQLHENV    henv);
```

Table 96 shows parameters for this function.

Table 96. 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.

The return codes are SQL_SUCCESS, SQL_ERROR, and SQL_INVALID_HANDLE.

6.5.8 Introduction to a CLI example

To illustrate the use of CLI interface, we modified the example that we used in 6.3.6, “Source code for SQLEMBED program” on page 336. Embedded SQL statements are replaced with subprocedures that call CLI functions.

All subprocedure prototypes required for this example are made available through the include file CLIPROTO. This file contains only selected functions that we used in our example. It also contains definitions of constants for CLI return codes, connect attributes, SQL data types, and other options used by CLI functions.

6.5.8.1 Source code for the SQLCLI program

The logic of the program SQLCLI is the same as that of the programs that we used in 6.3.6, “Source code for SQLEMBED program” on page 336, and 6.4.4, “A stored procedure example” on page 342. We wanted to demonstrate that the same functionality can be achieved by using either embedded SQL, stored procedures, or DB2 Call Level Interface. The program uses the display file DSPFIL1 to communicate with the end user.

To create this program, two steps are required:

1. Create the module using either PDM option 15 or CL command `CRTRPGMOD`
2. Create the program using either PDM option 26 or CL command `CRTPGM` with the parameter `BNDSRVPGM(QSYS/QSQCLI)`

```
 *********************************************************************
 *  Filename 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)
 *********************************************************************
 * Display file with subfile
FDspFil1   CF   E             WorkStn IndDS(DispInd)
F                                     SFile(SflRec1:RecNum)
 *-------------------------------------------------------------------
D RecNum          S              3  0
 * Indicator data structure for display file indicators
 *
D DispInd         DS
D  Exit                  3      3N
D  ListAllRec            4      4N
D  InsertRec             5      5N
D  UpdateRec             6      6N
D  DeleteRec             7      7N
D  Cancel               12     12N
D  InvalidRec           55     55N
D  ClearSfl             66     66N
 * Include copy member with CLI prototypes and SQL variables
 *
D/COPY RPGISCOOL/DBSRC,CLIPROTO                                        1
 * Program variables used by CLI interface
 *
D SQL_RC          S             10I 0                                 2
D henv            S             10I 0
D hdbc            S             10I 0
D DBName          S             18A
D DBNameP         S               *   INZ(%ADDR(DBName))
D DBUser          S             10A
D DBUserP         S               *   INZ(%ADDR(DBUser))
D DBPwd           S             10A
D DBPwdP          S               *   INZ(%ADDR(DBPwd))
D ConnOpt         S              5I 0
D IslLvl          S             10I 0
D IslLvlP         S               *   INZ(%ADDR(IslLvl))
D hstmt           S             10I 0
D SQLStm          S            128A
D SQLStmP         S               *   INZ(%ADDR(SQLStm))
D SQLState        S              6A
D SQLStateP       S               *   INZ(%ADDR(SQLState))
```

```
D SQLCode         S             10I 0
D SQLMsg          S             71A
D SQLMsgP         S               *   INZ(%ADDR(SQLMsg))
D MsgLenMax       S              5I 0
D MsgLen          S              5I 0
D ZeroBin         S            128A   INZ(*ALLX'00')
D ParamLen        S             10I 0
D ParamDec        S              5I 0
D pcbValue        S             10I 0 INZ(0)
D Param1P         S               *
D Param2P         S               *
D Param3P         S               *
D Param4P         S               *
D Param5P         S               *
 *--------------------------------------------------------------------
 * Begin of CLI itialization tasks
 *
C                   Eval      MsgLenMax=SQL_MAXMSG+1                  3
 * Allocate environment handle
C                   Eval      SQL_RC=SQLAlcEnv(henv)                   4
 * Allocate connection handle
C                   Eval      SQL_RC=SQLAlcCon(henv:hdbc)              5
 * Set connection options
C                   Eval      ConnOpt=SQL_ISOLVL                       6
C                   Eval      IslLvl=SQL_NONE
C                   Eval      SQL_RC=SQLSetCnOp(hdbc:ConnOpt:IslLvlP)
 * Connect to database
C                   Eval      DBName='AS25'                            7
C                   Eval      DBName=(%TRIM(DBName))+Zerobin
C                   Eval      DBUser='username'
C                   Eval      DBUser=(%TRIM(DBUser))+Zerobin
C                   Eval      DBPwd='password'
C                   Eval      DBPwd=(%TRIM(DBPwd))+Zerobin
C                   Eval      SQL_RC=SQLConnect(hdbc:DBNameP:SQL_NTS:
C                             DBUserP:SQL_NTS:DBPwdP:SQL_NTS)
 * Check CLI Initialization
C                   If        SQL_RC<>SQL_OK                           8
C                   Eval      SQL_RC=SQLError(henv:hdbc:hstmt:SQLStateP
C                             SQLCode:SQLMsgP:MsgLenMax:MsgLen)
C                   EndIf
 * End of CLI itialization tasks
 *--------------------------------------------------------------------
C                   Exfmt     DspRec1
 * Loop begin
C                   DoW       Not (Exit Or Cancel)
C                   Eval      InvalidRec=*Off
C                   Select
 * * * Insert record
C                   When      InsertRec
C                   Clear                   DspDes
C                   Clear                   DspQty
C                   Clear                   DspPrc
C                   Clear                   DspDat
C                   Eval      DspNum=PartNo
C                   Exfmt     DspRec2
 * Allocate statement handle for INSERT
C                   Eval      SQL_RC=SQLAlcStmt(hdbc:hstmt)           9
 * Bind value for 1. parameter marker PartNum
C                   Eval      Param1P=%ADDR(DspNum)                   10
C                   Eval      ParamLen=5
C                   Eval      ParamDec=0
C                   Eval      SQL_RC=SQLBindPar(hstmt:1:SQL_DECIM:
C                             SQL_DECIM:ParamLen:ParamDec:Param1P:
C                             pcbValue)
 * Bind value for 2. parameter marker PartDes
C                   Eval      Param2P=%ADDR(DspDes)                   10
C                   Eval      ParamLen=25
C                   Eval      ParamDec=0
C                   Eval      SQL_RC=SQLBindPar(hstmt:2:SQL_CHAR:
C                             SQL_CHAR:ParamLen:ParamDec:Param2P:
C                             pcbValue)
 * Bind value for 3. parameter marker PartQty
C                   Eval      Param3P=%ADDR(DspQty)                   10
C                   Eval      ParamLen=5
C                   Eval      ParamDec=0
C                   Eval      SQL_RC=SQLBindPar(hstmt:3:SQL_DECIM:
C                             SQL_DECIM:ParamLen:ParamDec:Param3P:
C                             pcbValue)
```

```
 * Bind value for 4. parameter marker PartPrc
C                   Eval      Param4P=%ADDR(DspPrc)                     10
C                   Eval      ParamLen=6
C                   Eval      ParamDec=2
C                   Eval      SQL_RC=SQLBindPar(hstmt:4:SQL_DECIM:
C                             SQL_DECIM:ParamLen:ParamDec:Param4P:
C                             pcbValue)
 * Bind value for 5. parameter marker PartDat
C                   Eval      Param5P=%ADDR(DspDat)                     10
C                   Eval      ParamLen=10
C                   Eval      ParamDec=0
C                   Eval      SQL_RC=SQLBindPar(hstmt:5:SQL_CHAR:
C                             SQL_DATE:ParamLen:ParamDec:Param5P:
C                             pcbValue)
 * Define and execute INSERT statement
C                   Eval      SQLStm='INSERT INTO PARTS '+               11
C                             'VALUES(?,?,?,?,?)'
C                   Eval      SQLStm=(%TRIM(SQLStm))+Zerobin
C                   Eval      SQL_RC=SQLExecDir(hstmt:SQLStmP:SQL_NTS)
 * Check for duplicate record error
C                   If        SQL_RC<>SQL_OK
C                   Eval      InvalidRec=*On
C                   Endif
 * Free statement handle for INSERT
C                   Eval      SQL_RC=SQLFreeStm(hstmt:SQL_DROP)         14
 * * * Display all records
C                   When      ListAllRec
 * Clear subfile
C                   Eval      RecNum=0
C                   Eval      ClearSfl=*On
C                   Write     SflRec2
C                   Eval      ClearSfl=*Off
 * Allocate statement handle for SELECT all records
C                   Eval      SQL_RC=SQLAlcStmt(hdbc:hstmt)              9
 * Define and execute SELECT statement
C                   Eval      SQLStm='SELECT * FROM PARTS ' +           11
C                             'ORDER BY PARTNUM'
C                   Eval      SQLStm=(%TRIM(SQLStm))+Zerobin
C                   Eval      SQL_RC=SQLExecDir(hstmt:SQLStmP:SQL_NTS)
 * Bind 1. column to program variable DspNum
C                   Eval      Param1P=%ADDR(DspNum)                     12
C                   Eval      ParamLen=1280
C                   Eval      SQL_RC=SQLBindCol(hstmt:1:SQL_DECIM:
C                             Param1P:ParamLen:pcbValue)
 * Bind 2. column to program variable DspDes
C                   Eval      Param2P=%ADDR(DspDes)                     12
C                   Eval      ParamLen=25
C                   Eval      SQL_RC=SQLBindCol(hstmt:2:SQL_CHAR:
C                             Param2P:ParamLen:pcbValue)
 * Bind 3. column to program variable DspQty
C                   Eval      Param3P=%ADDR(DspQty)                     12
C                   Eval      ParamLen=1280
C                   Eval      SQL_RC=SQLBindCol(hstmt:3:SQL_DECIM:
C                             Param3P:ParamLen:pcbValue)
 * Bind 4. column to program variable DspPrc
C                   Eval      Param4P=%ADDR(DspPrc)                     12
C                   Eval      ParamLen=1538
C                   Eval      SQL_RC=SQLBindCol(hstmt:4:SQL_DECIM:
C                             Param4P:ParamLen:pcbValue)
 * Bind 5. column to program variable DspDat
C                   Eval      Param5P=%ADDR(DspDat)                     12
C                   Eval      ParamLen=10
C                   Eval      SQL_RC=SQLBindCol(hstmt:5:SQL_DATE:
C                             Param5P:ParamLen:pcbValue)
 * Execute Fetch in loop to read all records
C                   DoU       SQL_RC=SQL_NODATA
C                   Eval      SQL_RC=SQLFetch(hstmt)                    13
C                   If        SQL_RC=SQL_NODATA
C                   Leave
C                   EndIf
C                   Eval      RecNum=RecNum+1
C                   Write     SflRec1
C                   EndDo
 * Free statement handle for SELECT all records
C                   Eval      SQL_RC=SQLFreeStm(hstmt:SQL_DROP)         14
C                   Exfmt     SflRec2
 * * * Display single record
C                   Other
```

```
 * Allocate statement handle for SELECT single record            9
C                   Eval      SQL_RC=SQLAlcStmt(hdbc:hstmt)
 * Bind value for 1. parameter marker PartNum
C                   Eval      Param1P=%ADDR(PartNo)              10
C                   Eval      ParamLen=5
C                   Eval      ParamDec=0
C                   Eval      SQL_RC=SQLBindPar(hstmt:1:SQL_DECIM:
C                             SQL_DECIM:ParamLen:ParamDec:Param1P:
C                             pcbValue)
 * Define and execute SELECT statement
C                   Eval      SQLStm='SELECT PARTDES, PARTQTY, ' +   11
C                             'PARTPRC, PARTDAT FROM PARTS ' +
C                             'WHERE PARTNUM=?'
C                   Eval      SQLStm=(%TRIM(SQLStm))+Zerobin
C                   Eval      SQL_RC=SQLExecDir(hstmt:SQLStmP:SQL_NTS)
 * Bind 1. column to program variable DspDes
C                   Eval      Param2P=%ADDR(DspDes)              12
C                   Eval      ParamLen=25
C                   Eval      SQL_RC=SQLBindCol(hstmt:1:SQL_CHAR:
C                             Param2P:ParamLen:pcbValue)
 * Bind 2. column to program variable DspQty
C                   Eval      Param3P=%ADDR(DspQty)              12
C                   Eval      ParamLen=1280
C                   Eval      SQL_RC=SQLBindCol(hstmt:2:SQL_DECIM:
C                             Param3P:ParamLen:pcbValue)
 * Bind 3. column to program variable DspPrc
C                   Eval      Param4P=%ADDR(DspPrc)              12
C                   Eval      ParamLen=1538
C                   Eval      SQL_RC=SQLBindCol(hstmt:3:SQL_DECIM:
C                             Param4P:ParamLen:pcbValue)
 * Bind 4. column to program variable DspDat
C                   Eval      Param5P=%ADDR(DspDat)              12
C                   Eval      ParamLen=10
C                   Eval      SQL_RC=SQLBindCol(hstmt:4:SQL_DATE:
C                             Param5P:ParamLen:pcbValue)
 * Execute Fetch to read single record
C                   Eval      SQL_RC=SQLFetch(hstmt)             13
C                   If        SQL_RC=SQL_NODATA
C                   Eval      InvalidRec=*On
C                   EndIf
 * Free statement handle for SELECT single record
C                   Eval      SQL_RC=SQLFreeStm(hstmt:SQL_DROP)  14
 * Check for no record found error
C                   If        Not InvalidRec
C                   Eval      DspNum=PartNo
C                   Exfmt     DspRec2
C                   Select
C                   When      Exit Or Cancel
C                   Leave
 * * * Update record
C                   When      UpdateRec
 * Allocate statement handle for UPDATE
C                   Eval      SQL_RC=SQLAlcStmt(hdbc:hstmt)       9
 * Bind value for 1. parameter marker PartDes
C                   Eval      Param1P=%ADDR(DspDes)              10
C                   Eval      ParamLen=25
C                   Eval      ParamDec=0
C                   Eval      SQL_RC=SQLBindPar(hstmt:1:SQL_CHAR:
C                             SQL_CHAR:ParamLen:ParamDec:Param1P:
C                             pcbValue)
 * Bind value for 2. parameter marker PartQty
C                   Eval      Param2P=%ADDR(DspQty)              10
C                   Eval      ParamLen=5
C                   Eval      ParamDec=0
C                   Eval      SQL_RC=SQLBindPar(hstmt:2:SQL_DECIM:
C                             SQL_DECIM:ParamLen:ParamDec:Param2P:
C                             pcbValue)
 * Bind value for 3. parameter marker PartPrc
C                   Eval      Param3P=%ADDR(DspPrc)              10
C                   Eval      ParamLen=6
C                   Eval      ParamDec=2
C                   Eval      SQL_RC=SQLBindPar(hstmt:3:SQL_DECIM:
C                             SQL_DECIM:ParamLen:ParamDec:Param3P:
C                             pcbValue)
 * Bind value for 4. parameter marker PartDat
C                   Eval      Param4P=%ADDR(DspDat)              10
C                   Eval      ParamLen=10
C                   Eval      ParamDec=0
```

```
C                   Eval      SQL_RC=SQLBindPar(hstmt:4:SQL_CHAR:
C                             SQL_DATE:ParamLen:ParamDec:Param4P:
C                             pcbValue)
 * Bind value for 5. parameter marker PartNum
C                   Eval      Param5P=%ADDR(DspNum)                   10
C                   Eval      ParamLen=5
C                   Eval      ParamDec=0
C                   Eval      SQL_RC=SQLBindPar(hstmt:5:SQL_DECIM:
C                             SQL_DECIM:ParamLen:ParamDec:Param5P:
C                             pcbValue)
 * Define and execute UPDATE statement
C                   Eval      SQLStm='UPDATE PARTS SET PARTDES=?, '+  11
C                             'PARTQTY=?, PARTPRC=?, PARTDAT=? '+
C                             'WHERE PARTNUM=?'
C                   Eval      SQLStm=(%TRIM(SQLStm))+Zerobin
C                   Eval      SQL_RC=SQLExecDir(hstmt:SQLStmP:SQL_NTS)
 * Free statement handle for UPDATE
C                   Eval      SQL_RC=SQLFreeStm(hstmt:SQL_DROP)       14
 * * * Delete record
C                   When      DeleteRec
 * Allocate statement handle for DELETE
C                   Eval      SQL_RC=SQLAlcStmt(hdbc:hstmt)            9
 * Bind value for 1. parameter marker PartNum
C                   Eval      Param1P=%ADDR(PartNo)                    10
C                   Eval      ParamLen=5
C                   Eval      ParamDec=0
C                   Eval      SQL_RC=SQLBindPar(hstmt:1:SQL_DECIM:
C                             SQL_DECIM:ParamLen:ParamDec:Param1P:
C                             pcbValue)
 * Define and execute DELETE statement                                 11
C                   Eval      SQLStm='DELETE FROM PARTS WHERE PARTNUM=?
C                   Eval      SQLStm=(%TRIM(SQLStm))+Zerobin
C                   Eval      SQL_RC=SQLExecDir(hstmt:SQLStmP:SQL_NTS)
 * Free statement handle for DELETE
C                   Eval      SQL_RC=SQLFreeStm(hstmt:SQL_DROP)         14
C                   EndSl
C                   EndIf
C                   EndSl
C                   Exfmt     DspRec1
 * Loop end
C                   EndDo
 *---------------------------------------------------------------------
 * Begin of CLI Termination tasks
 * Disconnect from database
C                   Eval      SQL_RC=SQLDisconn(hdbc)                 15
 * Free connection handle
C                   Eval      SQL_RC=SQLFreeCon(hdbc)                 16
 * Free environment handle
C                   Eval      SQL_RC=SQLFreeEnv(henv)                 17
 * End of CLI Termination tasks
 *---------------------------------------------------------------------
C                   Eval      *INLR = *On
```

SQLCLI program notes

1 Copy the source member CLIPROTO to include all required subprocedure prototypes.

2 Variables 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 will never be longer than this.

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 identified by the input environment handle.

6 The function SQLSetConnectOption() sets connection attributes for a particular connection. With SQL_NONE, we define that commitment control will not be used.

7 The function SQLConnect() establishes a connection to the target database, which must already be defined on the system for the connect to work. You can use the Work with Relational Database Directory Entries (WRKRDBDIRE) command to determine which databases have been defined already and to optionally define additional one.

8 An example how to check the success of executed CLI function. SQLError() returns the diagnostic information 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 specified by the connection handle. In our example, we use only one statement handle which is allocated to different SQL statements. Another possibility would be 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 to be substituted when the statement is executed. 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 executed. 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 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 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 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.

6.5.8.2 Source code for prototypes CLIPROTO

This source member is copied into the previous SQLCLI program. It contains subprocedure prototypes for CLI functions used by this program.

```
*******************************************************************
 * Filename CLIPROTO from DBSRC in RPGISCOOL
 * Function Prototype ALLOCATE ENVIRONMENT HANDLE
 *
 *          SQLRETURN  SQLAllocEnv (SQLHENV  *phenv);
 *******************************************************************
 * Return value = 0 (OK) or -1 (error)
DSQLAlcEnv        PR            10I 0 ExtProc('SQLAllocEnv')
 * Environmental handle
D                               10I 0
 *******************************************************************
 * Function Prototype ALLOCATE CONNECTION HANDLE
 *
 *          SQLRETURN  SQLAllocConnect (SQLHENV   henv,
 *                                      SQLHDBC  *phdbc);
 *******************************************************************
 * Return value = 0 (OK) or -1 (error) or -2 (invalid handle)
DSQLAlcCon        PR            10I 0 ExtProc('SQLAllocConnect')
 * Environmental handle
D                               10I 0 VALUE
 * Connection handle
D                               10I 0
 *******************************************************************
 * 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)
DSQLConnect       PR            10I 0 ExtProc('SQLConnect')
 * Connection handle
D                               10I 0 VALUE
 * Pointer of the field containing the name of the database
D                                 *   VALUE
 * Length of the name of the database
D                                5I 0 VALUE
 * Pointer of the field containing the user identification
D                                 *   VALUE
 * Length of the user identification
D                                5I 0 VALUE
 * Pointer of the field containing the password
D                                 *   VALUE
 * Length of the password
D                                5I 0 VALUE
 *******************************************************************
 * 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)
DSQLSetCnOp       PR            10I 0 ExtProc('SQLSetConnectOption')
 * Connection handle
D                               10I 0 VALUE
 * Connect option
D                                5I 0 VALUE
 * Pointer to the field containing the value of the connect option
D                                 *   VALUE
 *******************************************************************
 * Function Prototype ALLOCATE STATEMENT HANDLE
 *
 *          SQLRETURN SQLAllocStmt (SQLHDBC   hdbc,
 *                                  SQLHSTMT *phstmt);
 *******************************************************************
 * Return value = 0 (OK) or -1 (error) or -2 (invalid handle)
DSQLAlcStmt       PR            10I 0 ExtProc('SQLAllocStmt')
 * Connection handle
D                               10I 0 VALUE
 * Handle of the SQL statement
D                               10I 0
 *******************************************************************
 * 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)
DSQLPrepare       PR            10I 0 ExtProc('SQLPrepare')
 * Handle of the SQL statement
D                               10I 0 VALUE
 * Pointer to the field containing the SQL statement
D                                 *   VALUE
```

```
 * Length of the SQL statement
D                                5I 0 VALUE
 *****************************************************************
 * Function Prototype BIND BUFFER TO A PARAMETER MARKER
 *
 *          SQLRETURN  SQLBindParam(SQLHSTMT     hstmt,
 *                                  SQLSMALLINT  iparm,
 *                                  SQLSMALLINT  iType,
 *                                  SQLSMALLINT  pType,
 *                                  SQLINTEGER   pLen,
 *                                  SQLSMALLINT  pScale,
 *                                  SQLPOINTER   pData,
 *                                  SQLINTEGER  *pcbValue);
 *****************************************************************
 * Return value = 0 or 1 (OK) or -1 (error) or -2 (invalid handle)
DSQLBindPar       PR            10I 0 ExtProc('SQLBindParam')
 * Handle of the SQL statement
D                               10I 0 VALUE
 * Sequential parameter marker number
D                                5I 0 VALUE
 * Data type of the parameter (application)
D                                5I 0 VALUE
 * Data type of the parameter (SQL)
D                                5I 0 VALUE
 * Length of the parameter
D                               10I 0 VALUE
 * Decimal number of the parameter
D                                5I 0 VALUE
 * Pointer to the buffer containing the parameter
D                                 *   VALUE
 * Length of the parameter (se alfanumerico) or 0
D                               10I 0
 *****************************************************************
 * 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)
DSQLBindCol       PR            10I 0 ExtProc('SQLBindCol')
 * Handle of the SQL statement
D                               10I 0 VALUE
 * Sequential parameter marker number
D                                5I 0 VALUE
 * Data type of the parameter (application)
D                                5I 0 VALUE
 * Pointer to the program variable
D                                 *   VALUE
 * Length of the variable
D                               10I 0 VALUE
 * Length of the parameter
D                               10I 0
 *****************************************************************
 * 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)
DSQLExecute       PR            10I 0 ExtProc('SQLExecute')
 * Handle of the SQL statement
D                               10I 0 VALUE
 *****************************************************************
 * 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)
DSQLExecDir       PR            10I 0 ExtProc('SQLExecDirect')
 * Handle of the SQL statement
D                               10I 0 VALUE
```

```
 * Pointer of the field containing the SQL statement
D                                  *   VALUE
 * Length of the SQL statement
D                                 5I 0 VALUE
 *********************************************************************
 * 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)
DSQLFetch         PR            10I 0 ExtProc('SQLFetch')
 * Statement handle
D                                10I 0 VALUE
 *********************************************************************
 * Function Prototype LAST TRANSACTION
 *
 *          SQLRETURN  SQLTransact (SQLHENV      henv,
 *                                  SQLHDBC      hdbc,
 *                                  SQLSMALLINT  fType);
 *********************************************************************
 * Return value = 0 (OK) or -1 (error) or -2 (invalid handle)
DSQLTrans         PR            10I 0 ExtProc('SQLTransact')
 * Environmental handle
D                                10I 0 VALUE
 * Connection handle
D                                10I 0 VALUE
 * Action of last transaction: 0=COMMIT, 1=ROLLBACK
D                                 5I 0 VALUE
 *********************************************************************
 * 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)
DSQLError         PR            10I 0 ExtProc('SQLError')
 * Environmental handle
D                                10I 0 VALUE
 * Connection handle
D                                10I 0 VALUE
 * Handle of the SQL statement
D                                10I 0 VALUE
 * Pointer to the field that must contain the SQLSTATE
D                                  *   VALUE
 * SQLCODE returned from the database
D                                10I 0
 * Pointer to the field that must contain the error message
D                                  *   VALUE
 * Maximum length of the error message
D                                 5I 0 VALUE
 * Total length of the error message
D                                 5I 0
 *********************************************************************
 * 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)
DSQLFreeStm       PR            10I 0 ExtProc('SQLFreeStmt')
 * Handle of the SQL statement
D                                10I 0 VALUE
 * Mode of deallocation
D                                 5I 0 VALUE
 *********************************************************************
 * Function Prototype DISCONNECTION OF A DATABASE
 *
 *          SQLRETURN  SQLDisconnect (SQLHDBC  hdbc);
 *
 *********************************************************************
 * Return value = 0 or 1 (OK) or -1 (error) or -2 (invalid handle)
DSQLDisconn       PR            10I 0 ExtProc('SQLDisconnect')
 * Connection handle
D                               10I 0 VALUE
 *********************************************************************
 * Function Prototype CONNECTION DEALLOCATION HANDLE
 *
 *          SQLRETURN  SQLFreeConnect (SQLHDBC hdbc);
 *********************************************************************
 * Return value = 0 (OK) or -1 (error) or -2 (invalid handle)
DSQLFreeCon       PR            10I 0 ExtProc('SQLFreeConnect')
 * Connection handle
D                               10I 0 VALUE
 *********************************************************************
 * Function Prototype DEALLOCATION ENVIRONMENTAL HANDLE
 *
 *          SQLRETURN  SQLFreeEnv (SQLHENV  henv);
 *********************************************************************
 * Return value = 0 (OK) or -1 (error) or -2 (invalid handle)
DSQLFreeEnv       PR            10I 0 ExtProc('SQLFreeEnv')
 * Environmental handle
D                               10I 0 VALUE
 *********************************************************************
 * RETCODE values
 *********************************************************************
DSQL_OK           C                   CONST(0)
DSQL_OK_INF       C                   CONST(1)
DSQL_NODATA       C                   CONST(100)
DSQL_NEEDAT       C                   CONST(99)
DSQL_ERROR        C                   CONST(-1)
DSQL_INVHAN       C                   CONST(-2)
 *********************************************************************
 * Valid values for connect attribute
 *********************************************************************
DSQL_AUTIPD       C                   CONST(10001)
DSQL_ISOLVL       C                   CONST(0)
DSQL_NONE         C                   CONST(1)
DSQL_CHANGE       C                   CONST(2)
DSQL_CS           C                   CONST(3)
DSQL_ALL          C                   CONST(4)
DSQL_RR           C                   CONST(5)
 *********************************************************************
 * SQLFreeStmt option values
 *********************************************************************
DSQL_CLOSE        C                   CONST(0)
DSQL_DROP         C                   CONST(1)
DSQL_UNBIND       C                   CONST(2)
DSQL_RESET        C                   CONST(3)
 *********************************************************************
 * SQLTransact option values
 *********************************************************************
DSQL_COMMIT       C                   CONST(0)
DSQL_ROLLBK       C                   CONST(1)
 *********************************************************************
 * Standard SQL data types
 *********************************************************************
DSQL_CHAR         C                   CONST(1)
DSQL_NUMER        C                   CONST(2)
DSQL_DECIM        C                   CONST(3)
DSQL_INTEG        C                   CONST(4)
DSQL_SMINT        C                   CONST(5)
DSQL_FLOAT        C                   CONST(6)
DSQL_REAL         C                   CONST(7)
DSQL_DOUBLE       C                   CONST(8)
DSQL_DATTIM       C                   CONST(9)
DSQL_VARCH        C                   CONST(12)
DSQL_GRAPH        C                   CONST(95)
DSQL_VARGR        C                   CONST(96)
DSQL_DATE         C                   CONST(91)
DSQL_TIME         C                   CONST(92)
DSQL_TIMEST       C                   CONST(93)
DSQL_CD_DAT       C                   CONST(1)
DSQL_CD_TIM       C                   CONST(2)
DSQL_CD_TST       C                   CONST(3)
DSQL_ALLTYP       C                   CONST(0)
 *********************************************************************
 * C data type to SQL data type mapping
 *********************************************************************
```

```
DSQL_C_CHAR        C                   CONST(1)
DSQL_C_LONG        C                   CONST(4)
DSQL_C_SHRT        C                   CONST(5)
DSQL_C_FLOT        C                   CONST(7)
DSQL_C_DOUB        C                   CONST(8)
DSQL_C_DTTM        C                   CONST(9)
 *****************************************************************
 * Generally useful constants
 *****************************************************************
 * Null Terminated String
DSQL_NTS           C                   CONST(-3)
DSQL_MAXMSG        C                   CONST(70)
DSQL_SQLSTS        C                   CONST(5)
```

Try it yourself

You can try this example by compiling the code from this section on your AS/400 system. Use the following commands to create the program:

```
CRTRPGMOD MODULE(RPGISCOOL/SQLCLI) SRCFILE(RPGISCOOL/DBSRC)
CRTPGM PGM(RPGISCOOL/SQLCLI) BNDSRVPGM(QSQCLI)
```

To run the program, enter the following command:

```
CALL PGM(RPGISCOOL/SQLCLI)
```

6.6 Trigger programs

A trigger is a set of actions that are 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 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 using the database.

- *Easier maintenance*

  If a business policy changes, it is necessary to change only the corresponding trigger program instead of each application program.

- *Improve performance in client/sever environment*

  All rules are run in the server before returning the result.

To use trigger support on the AS/400 system, you must create a trigger program and add it to a physical file.

6.6.1 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. Once 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 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. This allows the before trigger to be used 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, we use the Remove Physical File Trigger (RMVPFTRG) command. Once 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 associated with a file. Specify `TYPE(*TRG)` or `TYPE(*ALL)` to get this list.

6.6.2 Creating a trigger program

To provide desired trigger support, a trigger program must be written in any high-level language, SQL or CL. We use, of course, 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.

Trigger program input parameters are:

- Trigger buffer, which contains the information about the current change operation that invoked the trigger program
- Trigger buffer length

6.6.2.1 Trigger buffer field descriptions

The trigger buffer has two logical sections, a static and a 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 97 shows the content of a trigger buffer.

Table 97. 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
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

From	To	Type	Field description
*	*	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

6.6.2.2 Coding snippet TRGBUF example of input parameters

To define input parameters for a trigger program, you can copy the source member TRGBUF from the file QRPGLESRC in library QSYSINC. But, if you want to give longer and meaningful names to these fields, you have to define input data structure in your program.

The following code snippet illustrates how trigger program input parameters can be defined in an RPG IV program:

```
 *
 * Trigger buffer
 *
D TrgBuffer       DS                                                2
D  FileName                     10
D  LibName                      10
D  MbrName                      10
D  TrgEvent                      1
D  TrgTime                       1
D  CmtLevel                      1
D  Reserved1                     3
D  CCSID                        10I 0
D  Reserved2                     8
D  OrgRecOffset                 10I 0
D  OrgRecLength                 10I 0
D  OrgRecNulOff                 10I 0
D  OrgRecNulLen                 10I 0
D  NewRecOffset                 10I 0
D  NewRecLength                 10I 0
D  NewRecNulOff                 10I 0
D  NewRecNulLen                 10I 0
D  Reserved3                    16
D  OrgRecord                    47
D  OrgRecordNul                  5
D  NewRecord                    47
D  NewRecordNul                  5
 *
 * Overlay fields for original record
 *
D  PartNumOrg                    5S 0 Overlay(OrgRecord)            4
D  PartDesOrg                   25    Overlay(OrgRecord:*Next)
D  PartQtyOrg                    5P 0 Overlay(OrgRecord:*Next)
D  PartPrcOrg                    6P 2 Overlay(OrgRecord:*Next)
D  PartDatOrg                     D   Overlay(OrgRecord:*Next)
 *
 * Overlay fields for new record
 *
D  PartNumNew                    5S 0 Overlay(NewRecord)            4
D  PartDesNew                   25    Overlay(NewRecord:*Next)
D  PartQtyNew                    5P 0 Overlay(NewRecord:*Next)
D  PartPrcNew                    6P 2 Overlay(NewRecord:*Next)
D  PartDatNew                     D   Overlay(NewRecord:*Next)
 *
 * Trigger buffer length
 *
D TrgBufLen       S             10I 0                               3
 *
 * Entry parameter list for trigger program
 *
C     *Entry        Plist                                           1
```

```
C                   Parm                    TrgBuffer
C                   Parm                    TrgBufLen
 *                                                                    5
C                   Eval      Date=%Subst(TrgBuffer:OrgRecOffset+38:10)
```

TRGBUF snippet notes

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 standalone field, type integer.

4 Using the Overlay keyword, we can redefine original and new record fields in the trigger buffer to access particular fields from those records.

5 Record offset in combination with %SUBST built-in function can be also used to access particular fields from a trigger buffer.

6.6.2.3 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. We recommend that the trigger program be compiled 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. We recommend that 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 results 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 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.

6.6.2.4 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.

6.7 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. End users and application programmers call this a *transaction*.

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

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 logical units of work from a prior failure.

6.7.1 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 begin and end of commitment control, start commit cycle, or a commit and rollback operation. It also uses record before images when 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 environment, journal and journal receiver are automatically created as a part of the SQL collection. Any SQL table created inside the SQL collection is automatically journaled.

If the database file is created using a native interface (CRTPF), you must take care to activate journaling. Three steps are required:

1. Use the Create Journal Receiver (CRTJRNRCV) command to create a journal receiver object.
2. Use 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.

6.7.2 Using commitment control with RPG native file operations

To use commitment control in an ILE RPG program with native file operations, perform the following tasks:

1. On the system, follow these 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 (`STRCMTCTL`) and End Commitment Control (`ENDCMTCTL`).
2. In the ILE RPG program, follow these 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.

6.7.2.1 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.
- 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.

6.7.2.2 Commit and rollback operations

To indicate that a database file is to 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 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 will run under commitment control.

The COMMIT keyword parameter must be set prior to 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 C specification, you can use two operation codes at the end of related group of changes to denote the end of 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 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.

The following code snippet illustrates the use of commitment control in a program with native file operations:

```
FParts     UF   E           K Disk    Commit(ComitFlag)
FTrans     UF   E           K Disk    Commit(ComitFlag)
 *
 *  If ComitFlag = '1' the files are opened under commitment control,
 *  otherwise they are not.
 *
C     *Entry        Plist
C                   Parm                    ComitFlag
 *
 *  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')
 *
C                   Update(E) PartR
C                   Update(E) TranR
 *
C                   If        ComitFlag = '1'
 *
C                   If        %Error
C                   Rolbk
C                   Else
C                   Commit
C                   EndIf
 *
C                   EndIf
```

6.7.3 Using commitment control with embedded SQL

SQL provides similar functions to support commitment control using embedded the SQL statements COMMIT and ROLLBACK.

6.7.3.1 Starting commitment control

The commitment control environment for ILE RPG programs with embedded SQL statements is started automatically. DB2 for AS/400 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 (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.

6.7.3.2 Commit and rollback statements

In the program, we can either commit all transaction changes the using embedded SQL statement COMMIT or back out of these changes using the ROLLBACK statement.

Both statements have an optional HOLD parameter, which can be useful when using the cursor technique to fetch rows from result table. If HOLD is specified, currently open cursors are not closed, and all resources acquired during the unit of work are held. Locks on specific rows and objects 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.

6.7.4 Using commitment control with the CLI interface

The CLI interface provides support for commitment control by calling specific CLI functions, described earlier in 6.5, “Call Level Interface” on page 348.

6.7.4.1 Starting commitment control

Function SQLSetConnectionOption() sets connection attributes including transaction isolation level for commitment control. This function should be executed in the initialization phase of our CLI program to define the requested level of commitment control. The following connect options are supported:

1 SQL_COMMIT_NONE
2 SQL_COMMIT_CHG
3 SQL_COMMIT_CS
4 SQL_COMMIT_ALL
5 SQL_COMMIT_RR

This code snippet contains an example of calling SQLSetConnectionOption() to define the change isolation level for commitment control:

```
 * Variable definitions
D ConnOpt         S              5I 0
D IslLvl          S             10I 0
D IslLvlP         S               *   INZ(%ADDR(IslLvl))
 * Set connection options
C                   Eval      ConnOpt=SQL_ATTR_COMMIT
C                   Eval      IslLvl=SQL_COMMIT_CHG
C                   Eval      SQL_RC=SQLSetCnOp(hdbc:ConnOpt:IslLvlP)
```

SQL_ATTR_COMMIT is a numeric constant with a value of 0 and signals to CLI that we want to define option for commitment control.

SQL_COMMIT_CHG is another numeric constant with a value of 2 and defines the change isolation level.

6.7.4.2 The commit and rollback function

Another function, SQLTransact(), should be used to perform commit or rollback operations at the end of transaction. All changes to the database 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 invoke this function:

```
C                   Eval      SQL_RC=SQLTrans(henv:hdbc:SQL_COMMIT)
```

The numeric constant SQL_COMMIT with value 0 defines commit operation. Another constant SQL_ROLLBACK with value 1 represents rollback operation.

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 OS/400?

Consider the following IBM manuals for reference:

- *DB2 for OS/400 SQL Programming*, SC41-4611
- *DB2 for OS/400 SQL Reference*, SC41-4612
- *DB2 for OS/400 SQL Call Level Interface*, SC41-5806

Visit the following Web sites for programming samples and Frequently Asked Questions (FAQs) and their answers:

- `http://www.as400.ibm.com`
- `http://www.as400.ibm.com/db2/clifaq.htm`
- `http://www.as400.ibm.com/tstudio/db2_400/cli_rpg/rpg_intro.htm`
- `http://www.as400.ibm.com/tstudio/tech_ref/cli/cli1.htm`

Chapter 7. A modern tool for a modern language: CODE/400

While modernizing your programming language and style, as suggested in other parts of this book, perhaps it is also time to consider modernizing your development toolset. You may still be using mostly host-based tools, such as PDM and SEU, to enter and compile your source code. Instead, consider using a more modern and productive edit and compile platform based on a graphical workstation.

There are multiple workstation editors customized for AS/400 and RPG environments. As a way of illustrating the advantages to be gained from these types of tools, we consider the workstation development platform from IBM, called CODE/400 (CoOperative Development Environment).

CODE/400 is a toolset that contains multiple types of tools:

- Language-sensitive editor
- Code verifier
- DDS designer
- Debugger
- Project organizer

In this chapter, we concentrate primarily on the editor and verifier that are part of CODE/400, with a quick look at the DDS designer. The basic text-based tools for working with, editing, and compiling code and designing screens and reports (SEU, PDM, SDA and RLU) have not had significant enhancements in several releases. However, the workstation-based CODE/400 toolset continues to see significant enhancements regularly.

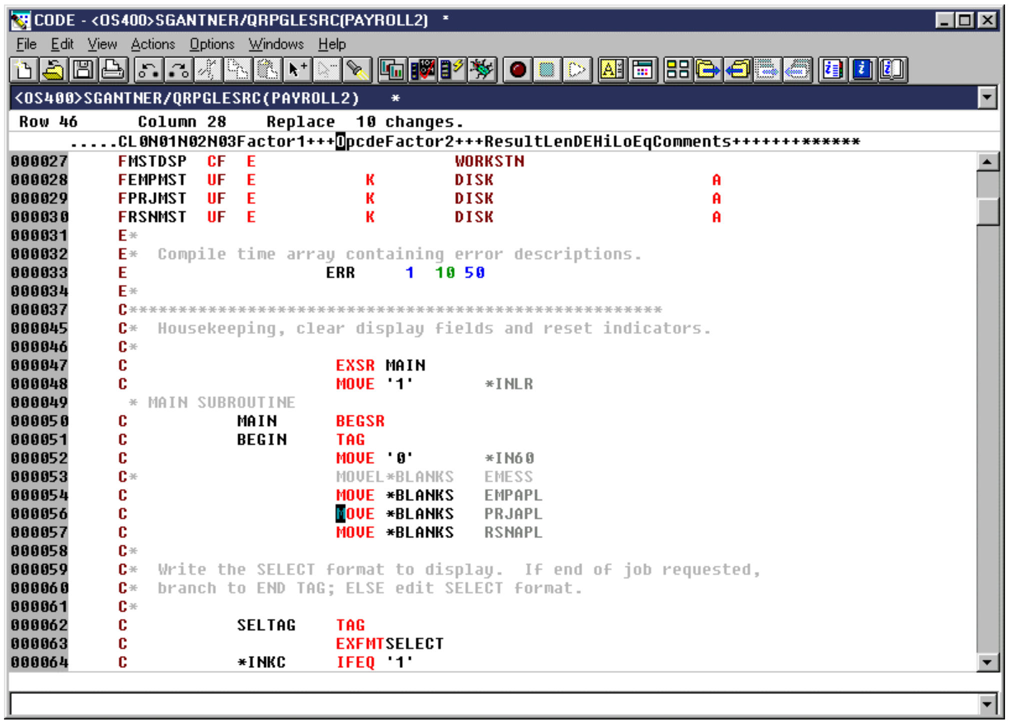
7.1 The CODE/400 editor

The editor that is part of the CODE/400 toolset is called Live Parsing EXtensible (LPEX) editor. It can be extended by adding user-written macros. However, many AS/400 programmers find the CODE/400 editor a productivity boon just as it comes, without user-written extensions.

Because the CODE/400 editor is workstation based, it takes advantage of workstation features not readily available to text-based editors, such as SEU. For example, a feature called *token highlighting* allows the CODE/400 editor to highlight and emphasize the different parts of each RPG specification by color coding the parts. For example, on the C specification, Factor 1 and Result may appear in gray, while the operation code is in red and Factor 2 is in black.

Perhaps the part of token highlighting that is most helpful is the fact that commented lines of code are in a different color from the rest of the code and, therefore, stand out immediately as comments. This can save hours of debugging time in cases where otherwise valid lines of code have been commented out, but, for example, the programmer that is debugging the code has not noticed the "*" in column 7.

Figure 42 on page 390 illustrates the color variation of the token highlighting. Notice how the code stands out and the comments fade.

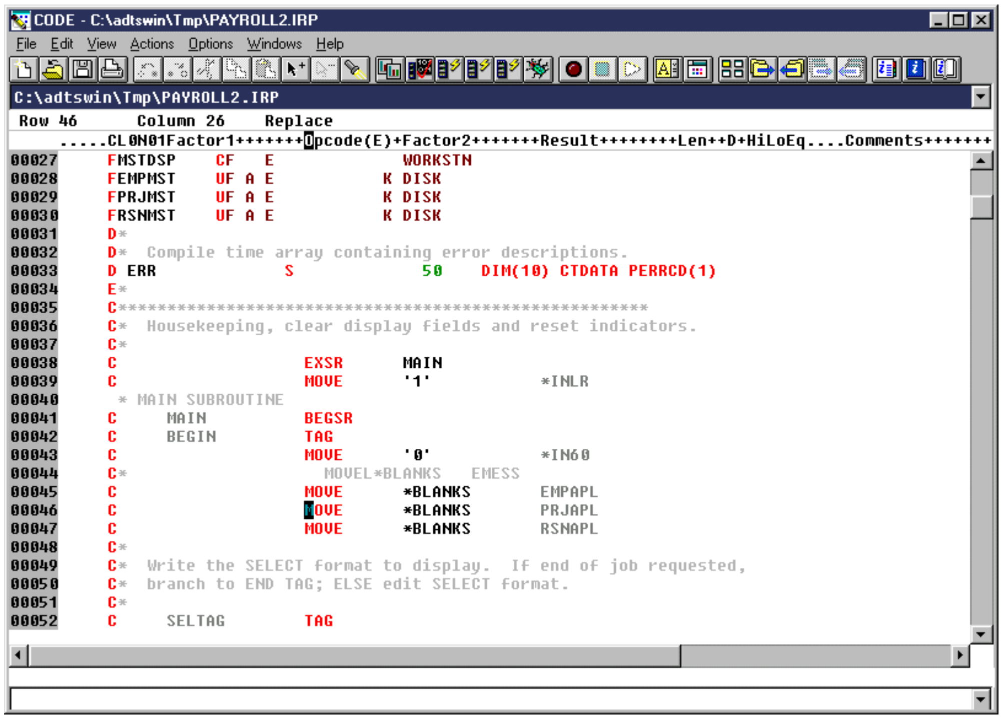

CODE - <OS400>SGANTNER/QRPGLESRC(PAYROLL2) *
File Edit View Actions Options Windows Help

```
<OS400>SGANTNER/QRPGLESRC(PAYROLL2)    *
Row 46       Column 28    Replace  10 changes.
.....CL0N01N02N03Factor1+++OpcdeFactor2+++ResultLenDEHiLoEqComments+++++++******
     FMSTDSP  CF  E                    WORKSTN
     FEMPMST  UF  E           K        DISK                      A
     FPRJMST  UF  E           K        DISK                      A
     FRSNMST  UF  E           K        DISK                      A
     E*
     E*  Compile time array containing error descriptions.
     E                    ERR     1  10 50
     E*
     C******************************************************
     C*  Housekeeping, clear display fields and reset indicators.
     C*
     C                     EXSR MAIN
     C                     MOVE '1'       *INLR
      * MAIN SUBROUTINE
     C           MAIN      BEGSR
     C           BEGIN     TAG
     C                     MOVE '0'       *IN60
     C*                    MOVEL*BLANKS   EMESS
     C                     MOVE *BLANKS   EMPAPL
     C                     MOVE *BLANKS   PRJAPL
     C                     MOVE *BLANKS   RSNAPL
     C*
     C*  Write the SELECT format to display.  If end of job requested,
     C*  branch to END TAG; ELSE edit SELECT format.
     C*
     C           SELTAG    TAG
     C                     EXFMTSELECT
     C           *INKC     IFEQ '1'
```

Figure 42. A sample CODE/400 editor window

Figure 42 shows RPG/400 code, and not RPG IV code. One reason this code was chosen for the first example is to illustrate that you can begin to use CODE/400 even if you have not yet moved all your code to RPG IV. CODE/400 supports many forms of RPG on the AS/400 system, including RPG, RPG36, RPG38, and SQLRPG. It also supports many other AS/400 language types, including CL, DDS, C, COBOL, and Java.

Another reason for choosing RPG/400 code in this example is to illustrate how CODE/400 can help in your conversion to RPG IV. From a single pull-down menu option, CODE/400 converts either a selected portion of the source or an entire source member to RPG IV. Using this option, you can easily convert source members visually and then save the resulting temporary workstation file back to an AS/400 source member. Alternatively, you can simply include logic from an existing RPG/400 program. For example, you can include it into a new RPG IV program or module by cutting and pasting the portion of the logic you need from the temporary file and subsequently discarding it.


CODE - C:\adtswin\Tmp\PAYROLL2.IRP
File Edit View Actions Options Windows Help
C:\adtswin\Tmp\PAYROLL2.IRP

```
Row 46      Column 26     Replace
     .....CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....Comments+++++++
00027     FMSTDSP    CF    E             WORKSTN
00028     FEMPMST    UF A E           K DISK
00029     FPRJMST    UF A E           K DISK
00030     FRSNMST    UF A E           K DISK
00031     D*
00032     D*  Compile time array containing error descriptions.
00033     D ERR             S             50    DIM(10) CTDATA PERRCD(1)
00034     E*
00035     C******************************************************
00036     C*  Housekeeping, clear display fields and reset indicators.
00037     C*
00038     C                   EXSR      MAIN
00039     C                   MOVE      '1'           *INLR
00040      * MAIN SUBROUTINE
00041     C     MAIN          BEGSR
00042     C     BEGIN         TAG
00043     C                   MOVE      '0'           *IN60
00044     C*                    MOVEL*BLANKS   EMESS
00045     C                   MOVE      *BLANKS       EMPAPL
00046     C                   MOVE      *BLANKS       PRJAPL
00047     C                   MOVE      *BLANKS       RSNAPL
00048     C*
00049     C*  Write the SELECT format to display.  If end of job requested,
00050     C*  branch to END TAG; ELSE edit SELECT format.
00051     C*
00052     C     SELTAG        TAG
```

Figure 43. Code sample converted to RPG IV

Figure 43 shows the results of converting the sample code to RPG IV. Notice that the results are the same as if we had issued a Convert RPG Source (CVTRPGSRC) command on the AS/400 host. However, in this case, CODE/400 has created a temporary file on the workstation. You can decide now whether to save the converted source to a member on the system or to simply copy part of the logic to another source member written in RPG IV.

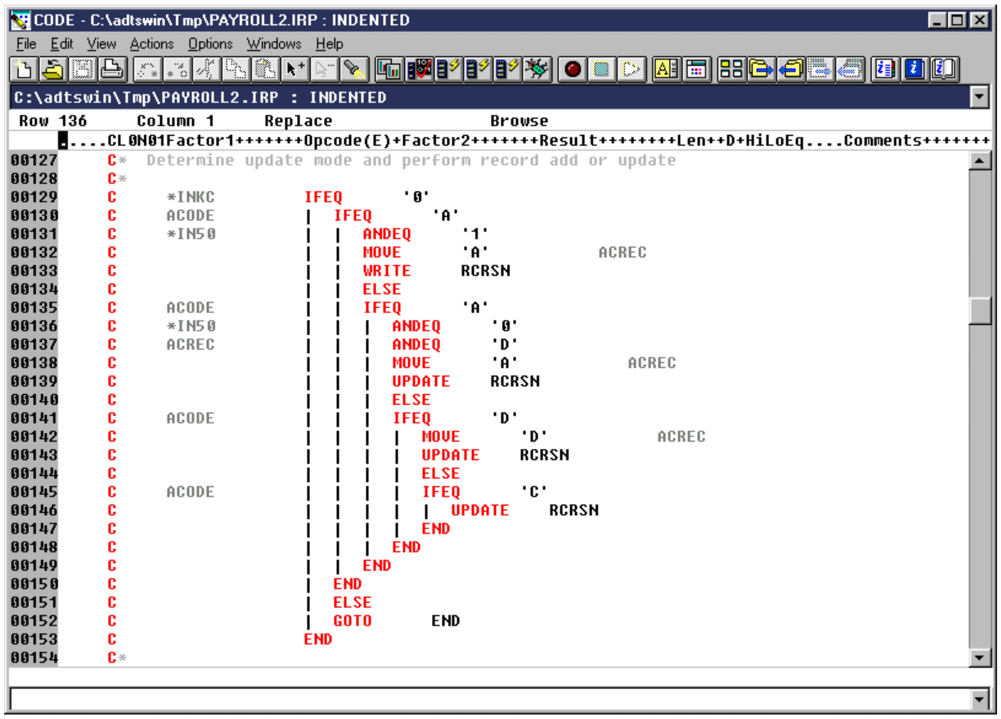
Note the source sequence numbers in Figure 43 as in the SEU editor. SEU line commands (such as C, A, B, I) can be used there, just as they can in SEU. However, there are also tools such as the more workstation-oriented cut, copy, and paste, as illustrated on the tool bar. For programmers who prefer not to use the SEU-type line commands, the sequence numbers can be removed from the edit window via an option from a menu.

Prompting on RPG source statements works much like it does in SEU by pressing F4. However, many programmers find it less necessary to prompt in the CODE/400 editor because the CODE/400 edit panel understands the parts of the RPG specification, and the Tab key is programmed to jump from "token" to "token". For example, on a C specification, the tab key allows programmers to jump quickly and easily from Factor 1 to Operation Code to Factor 2 to Result, etc. And the format line in the CODE/400 editor changes automatically to the format of the source line where the cursor is currently positioned. Therefore, it is easy to tell always exactly what part of the specification your cursor is on without prompting.

Other useful features of the editor not found in SEU include:

- Multiple levels of "undo" support
- Autosave of the source code to a local workstation file periodically

- A filter to see only the lines of code containing a specific string, such as a variable name
- An indented view of nested logic (illustrated in Figure 44)


CODE - C:\adtswin\Tmp\PAYROLL2.IRP : INDENTED
File Edit View Actions Options Windows Help

```
C:\adtswin\Tmp\PAYROLL2.IRP  : INDENTED
Row 136      Column 1      Replace                     Browse
 .....CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....Comments+++++++
     C*  Determine update mode and perform record add or update
     C*
     C      *INKC          IFEQ      '0'
     C      ACODE          |  IFEQ      'A'
     C      *IN50          |  |  ANDEQ     '1'
     C                     |  |  MOVE      'A'            ACREC
     C                     |  |  WRITE     RCRSN
     C                     |  |  ELSE
     C      ACODE          |  |  IFEQ      'A'
     C      *IN50          |  |  |  ANDEQ     '0'
     C      ACREC          |  |  |  ANDEQ     'D'
     C                     |  |  |  MOVE      'A'            ACREC
     C                     |  |  |  UPDATE    RCRSN
     C                     |  |  |  ELSE
     C      ACODE          |  |  |  IFEQ      'D'
     C                     |  |  |  |  MOVE      'D'            ACREC
     C                     |  |  |  |  UPDATE    RCRSN
     C                     |  |  |  |  ELSE
     C      ACODE          |  |  |  |  IFEQ      'C'
     C                     |  |  |  |  |  UPDATE    RCRSN
     C                     |  |  |  |  END
     C                     |  |  |  END
     C                     |  |  END
     C                     |  END
     C                     |  ELSE
     C                     |  GOTO      END
     C                     END
     C*
```

Figure 44. Sample code with indented view

7.2 The CODE/400 verifier

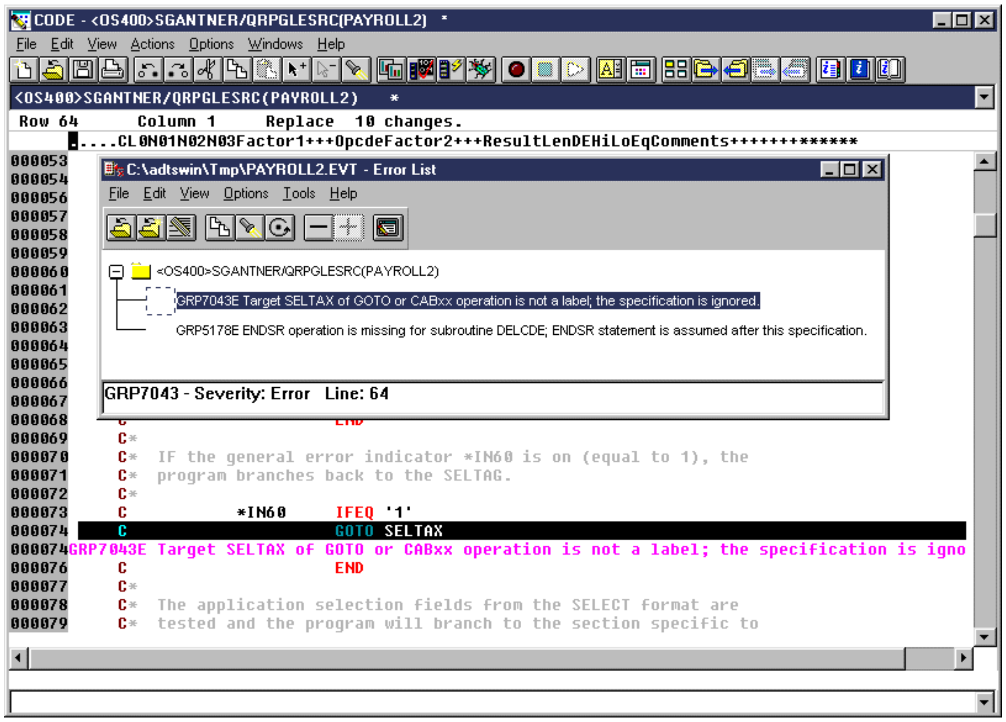
For many programmers, the best part of the CODE/400 toolset is the code verifier. The verifier verifies the validity of the code in the member. It not only checks the language syntax (which the editor also checks, as the SEU editor does), but it can also check the semantics of the code. It can check everything about the code that the compiler would check prior to creating a compiled program or module object. For example, the verifier can detect errors such as an RPG variable name that has not been defined or that has been misspelled.

The verifier runs on the programmer's workstation, but automatically communicates with the AS/400 system to obtain information needed from the host, such as external file descriptions or /COPY members still on the AS/400 system. When the code you have written or modified verifies without errors, it will almost certainly compile without errors on the AS/400 system. This local verification means there is no more need to submit multiple compile jobs to the AS/400 system to detect the kind of errors previously caught at compilation time.

The CODE/400 verifier has an option to *cache* (store a local workstation copy of) the external file descriptions and /COPY members used in a source member. This option reduces the number of times that the verifier needs to communicate with the AS/400 host system, and therefore improves the speed of the verification process.

Another big advantage of caching is that the programmer can continue to do work without any connection to the AS/400 host system. This makes it much easier for programmers who work as telecommuters from home or who simply occasionally find the need to finish some programming task away from the office. If the file descriptions are cached and the source members copied to the local drive on the workstation, programmers can continue to not only edit, but also verify their code in a disconnected state.

With the CODE/400 verifier, there is no need to look for your compile-type errors. Any errors on a code verification (or a compile, in case you did not verify first) pop up in a window over your editor panel. Then you can double-click on an error in the list, and the editor automatically is positioned to the line of code in error. This error feedback mechanism can save a significant amount of time in finding and correcting the kind of errors normally caught at compilation time. An example of the verifier and its error feedback window is in Figure 45.


CODE - <OS400>SGANTNER/QRPGLESRC(PAYROLL2) *
File Edit View Actions Options Windows Help
<OS400>SGANTNER/QRPGLESRC(PAYROLL2) *
C:\adtswin\Tmp\PAYROLL2.EVT - Error List
File Edit View Options Tools Help
<OS400>SGANTNER/QRPGLESRC(PAYROLL2)

```
Row 64       Column 1      Replace   10 changes.
 ....CL0N01N02N03Factor1+++OpcdeFactor2+++ResultLenDEHiLoEqComments++++++*******
```

```
GRP7043E Target SELTAX of GOTO or CABxx operation is not a label; the specification is ignored.
GRP5178E ENDSR operation is missing for subroutine DELCDE; ENDSR statement is assumed after this specification.
```

```
GRP7043 - Severity: Error   Line: 64
```

```
     C*
     C*  IF the general error indicator *IN60 is on (equal to 1), the
     C*  program branches back to the SELTAG.
     C*
     C             *IN60      IFEQ '1'
     C                        GOTO SELTAX
GRP7043E Target SELTAX of GOTO or CABxx operation is not a label; the specification is igno
     C                        END
     C*
     C*  The application selection fields from the SELECT format are
     C*  tested and the program will branch to the section specific to
```

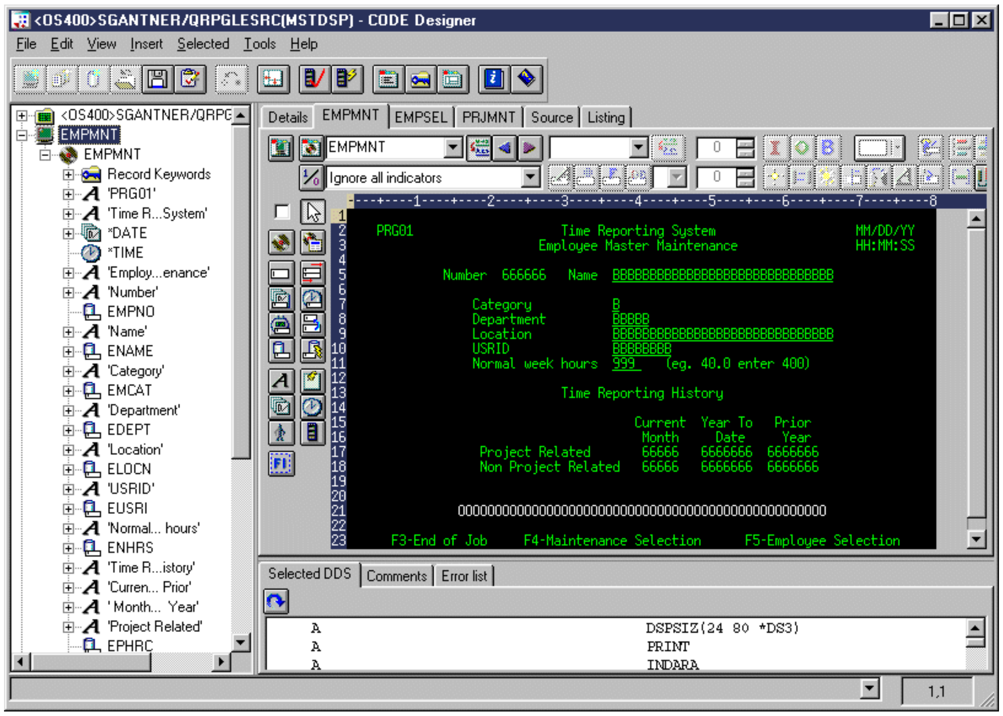
Figure 45. CODE/400 verifier error feedback

7.3 The CODE/400 Designer for DDS

To help programmers develop and maintain DDS for display files and printer files, CODE/400 has a tool called CODE Designer. In the latest release of CODE/400, the designer can also be used to create or maintain physical data file DDS.

The graphical capabilities of the workstation-based designer allow for significantly enhanced features over the use of Screen Design Aid (SDA) or Report Layout Utility (RLU). In addition, the same design tool is used for both display files and printer files, so you only need to learn one tool instead of two.

Since we are primarily interested in RPG coding, we do not go into detail about the CODE Designer. However, you can see a sample of the design panel for a display file in Figure 46.



```
-+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8

   PRG01                  Time Reporting System                    MM/DD/YY
                      Employee Master Maintenance                  HH:MM:SS

              Number  666666   Name  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBB

                  Category           B
                  Department         BBBBB
                  Location           BBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
                  USRID              BBBBBBBB
                  Normal week hours  999    (eg. 40.0 enter 400)

                              Time Reporting History

                                        Current  Year To   Prior
                                         Month     Date     Year
                   Project Related       66666   6666666  6666666
                   Non Project Related   66666   6666666  6666666


                 00000000000000000000000000000000000000000000000000

       F3-End of Job     F4-Maintenance Selection      F5-Employee Selection
```

```
A                                      DSPSIZ(24 80 *DS3)
A                                      PRINT
A                                      INDARA
```

Figure 46. A sample CODE Designer session for display file DDS

7.4 Other tools included with CODE/400

In addition to the editor, verifier, and designer tools we already looked at, CODE/400 includes a graphical interactive source view debugger and a project organizer, which is a list manager much like Programming Development Manager (PDM). CODE/400 comes packaged with VisualAge for RPG in a product called "VisualAge RPG and CODE/400" (5769-CL3). See Chapter 8, “VisualAge for RPG as a GUI for RPG applications”, for more information about the companion tool VisualAge for RPG.

7.5 More information about CODE/400

For more information about CODE/400, visit the IBM CODE/400 and VisualAge RPG Web site: `http://www.software.ibm.com/ad/varpg`

This Web site (as of this writing) offers an evaluation version of the product for you to download and try CODE/400 for yourself. The evaluation version has all the features of CODE/400, with two limitations. Service packs cannot be applied to the evaluation version, and a limited number of AS/400 host file opens are allowed with the evaluation version. If you do not want to download the code, you may sign up to have a CD sent to you containing the evaluation version.

In addition, you can download an interactive tutorial from this Web site. Many other helpful documents and references are also available.

Chapter 8. VisualAge for RPG as a GUI for RPG applications

VisualAge for RPG (VARPG) is a tool that can be used to create client/server applications on a Windows workstation. VARPG uses the RPG IV language in combination with a Graphical User Interface (GUI) design tool.

An application written with VARPG can reuse application code from the AS/400 host system. It can also easily access data from the AS/400 database using native RPG operations, such as CHAIN, READ, or UPDATE. In addition, calling RPG (or other language) programs on the AS/400 host can be accomplished by an ordinary program call with parameters, as on the AS/400 host.

In addition to running the resulting client application on a Windows 32-bit client, VARPG can now generate Java source code. The Java code can subsequently be run on virtually any type of client, as long as there is a Java Virtual Machine available on that client.

This chapter discusses some options for creating a thin user interface layer that may be suitable for using in conjunction with background RPG host-based application programs.

8.1 The different VARPG application models

There are two primary types of models for client/server applications: *thin client* and *thick client*. Using the thin client model is often preferable for ease of change management, more efficient utilization of system and network resources, and for increased reliability.

Thick client applications include almost all the application code in the workstation portion of the application, using the AS/400 server primarily for database access. VisualAge RPG (VARPG) allows programmers to create *thick client* applications.

Thin client applications separate the application logic between the workstation and the AS/400 server, leaving only the GUI handling portion of the application logic on the workstation. A large portion of the application logic, including most or all the database access logic, resides in server programs running on the AS/400 server.

These thin client applications are not to be confused with Thin Client hardware, called the IBM Network Station, which is a simple workstation without local persistent hardware storage. The term thin client application in this chapter is used for applications running on a Windows 95/NT workstation and exploiting the AS/400 server for major computing tasks.

A VARPG thick client application would do all processing including database access from the workstation and follow much the same programming style found in today's RPGIII or RPGIV applications. The only difference is that it runs on the workstation instead of the AS/400 system. File specifications are used to specify which database files to access and native RPG operation codes, such as READ, CHAIN, etc., are coded to access the data on the server. The AS/400 functions as a data server but does minimal computing to support the VARPG application in this thick client application model.

The disadvantage of this model is the limited capability of reusing modules and the increased management effort to provide change management for these applications. It also under utilizes the processing power of the server by moving processing onto client workstations.

On the other hand, by making the client portion of the application thinner, the amount of code running on the client is reduced. Consequently, the complexity of the application is reduced, and reusability is greatly enhanced. Maintenance of applications may likely be easier as well. This chapter discusses two possible implementations of the thin client application model in VARPG applications.

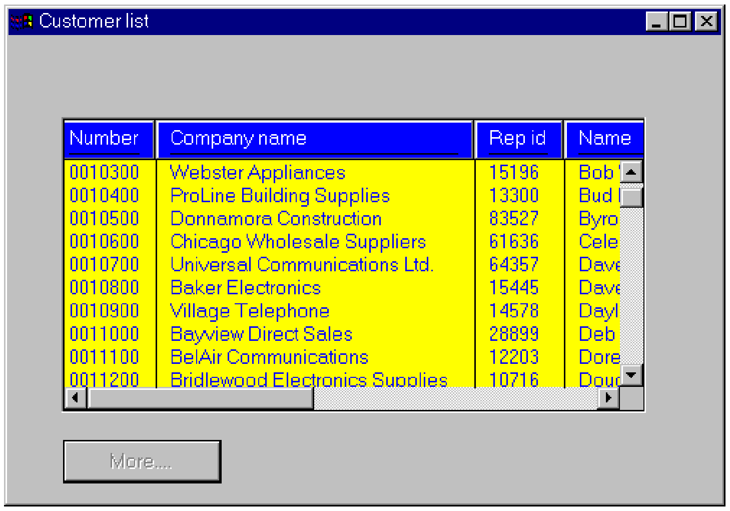
8.2 VARPG thin application models

The thin VARPG application model can be implemented in multiple ways. Two of these are described in this section. One implementation uses remote calls to an AS/400 system. The other utilizes data queues on the AS/400 system.

First, we look at the user interface. The same user interface is used in both examples.

8.3 The user interface for the client application

This is a simple application that reads data from a customer file and fills a subfile with 10 records at a time. Figure 47 shows the user interface.



Number	Company name	Rep id	Name
0010300	Webster Appliances	15196	Bob
0010400	ProLine Building Supplies	13300	Bud
0010500	Donnamora Construction	83527	Byro
0010600	Chicago Wholesale Suppliers	61636	Cele
0010700	Universal Communications Ltd.	64357	Dave
0010800	Baker Electronics	15445	Dave
0010900	Village Telephone	14578	Dayl
0011000	Bayview Direct Sales	28899	Deb
0011100	BelAir Communications	12203	Dore
0011200	Bridlewood Electronics Supplies	10716	Dou

Figure 47. VisualAge for RPG: Example user interface

It consists of a Window with Canvas, a subfile, and a push button to load an additional page of records into the subfile. The subfile size for this particular example is 10 records, but this can be changed by increasing the height of the subfile part.

The following VARPG part names are used in this example:

- **Window**: WIN1
- **Subfile**: SUB1
- **Pushbutton**: PSBMORE

8.4 Sample application using remote calls

In traditional AS/400 RPG programs, the user interface code and database access logic are intermixed into one module. Part of this structure stems from the history of RPG and the usage of the Original Program Model on the AS/400 system that forced this model onto the programmer to achieve good performance. The idea for the thin application model in VARPG is to split the user interface logic completely from the database access logic and run it on different systems. The user interface logic runs on a Windows client. The database access logic runs on the AS/400 server.

In this sample application, we show how to support reading records of data from the database and placing this data into a GUI subfile. The program on the AS/400 system can also support full database access (READ and WRITE). This can be implemented by supplying one program for each different access method or by passing the desired operations as parameters to a single server program.

8.4.1 The client program

The main part of the client side program is the user interface. It is created the same way as in all VARPG applications and can use the external data base descriptions of the AS/400 system by using database reference fields. Any validation checking specified in the database (for example, range or values checking specified in the DDS) is done automatically on the client by the VARPG runtime component.

The client program requests data from the server by calling a server data access program. The data itself is passed via parameters. The client program does not use file specifications. Instead, the data definition is done through externally described data structures. This way the programmer still gets the benefits of external field descriptions in their VARPG program.

8.4.2 The server program

Since the VARPG client program excludes the database access logic, this function is now provided in the server program. The AS/400 server program contains all file definitions and operations to handle database processing.

Data is exchanged by moving a data structure as a parameter between the client and the server program. The data structure contains the field definitions of the data file record format. In this example for accessing a collection of records, we used a multi-occurrence data structure. The number of occurrences is equal to the numbers of records to be passed. In this example, it is 10.

Any operational information (error information, for example) can be passed by a parameter as well.

The server program is invoked by the first call from the VARPG client program and ends after each request. The return operation code is used with the Last Record (LR) indicator set off in order to keep the invocation environment. This improves performance in subsequent calls since no initialization is needed and the database file does not need to be reopened. This also requires the program to be created to run in either the Default activation group or a Named ILE activation group. The use of a *NEW activation group would destroy the invocation environment and free storage immediately.

8.4.3 Sample RPG source for the client side

The VARPG program consists of the D and C specs (specifications). First, we look at the data definition (D) specifications:

- The D specs define the fields used as parameters:
  - Multi-occurrence structure Cust.
  - Numeric field CustElem contains the maximum number of records being requested.
  - Named indicator EOF is passed when the end of file indicator is set to ON in the server program.
  - Numeric field NRecords contains the number of records returned.
- Two working fields are specified:
  - FileEnd is a named indicator for keeping the file end condition.
  - Counter is a counter for the DO loop.
- One constant to define the program being called on the server:

  GetRec defines the linkage to the server and the actual name of the server program. Make sure the program name is specified in upper case in the Const keyword for GetRec.

```
D Cust          E Ds                 ExtName(Customer)
D                                    Occurs(10)
D                                    INZ
D EOF             S               N   INZ
D NRecords        S              2  0
D FileEnd         S               N   INZ
D Counter         S              2  0
D CustElem        S              2  0 INZ(%Elem(Cust))

D GetRec          C                  Linkage(*Server)
D                                    Const('GETREC')
```

- The C specs contain one Action Subroutine that is linked to three events:
  - Press event from the push button part PSBMore.
  - Create event from the window part Win1 (inked to PSBmore’s press action subroutine).
  - Page end event from the subfile part Sub1.

The first statement calls to the server program to retrieve more records.

The rest of the logic simply processes the data passed via parameter and moves it from the multi-occurrence data structure to the subfile.

After the subfile is filled with a set of records, the highest record number in the subfile is applied to the SetTop attribute to move this set of records into the visible area of the subfile.

Next, if the end of file is reached, the More push button is set to be disabled.

Be aware that the Page Down keys still work so it is possible to cause an event that will trigger this action subroutine even with a disabled push button.

```
C     PSBMore       BegAct    PRESS         Win1
C                   Call      GetRec
C                   Parm                    Cust
C                   Parm                    CustElem
```

```
C                   Parm      %EOF          EOF
C                   Parm                    NRecords
C                   Eval      Counter = 1

C                   Dow       Counter <= NRecords And Not FileEnd
C     Counter       Occur     Cust
C                   Write     Sub1
C                   Eval      Counter = Counter+1
C                   EndDo

C                   Eval      %SetAtr('Win1':'Sub1':'SetTop')
C                                 = %GetAtr('Win1':'Sub1':'Count')

C                   If        EOF
C                   Eval      %SetAtr('Win1':'PSBMore':'Enabled') = 0
C                   Eval      FileEnd = *On
C                   EndIf
      *
C                   EndAct
```

As you can see, the client end of this code is straight forward and minimizes the processing on the workstation.

8.4.4 Sample RPG source for the server side

The file specification defines the external database file named Customer. The data definition specifications define the parameters to be passed. These definitions must match the parameters defined in the client program earlier. Count represents a work variable for the counter in the DO loop. CustElem contains the number of elements in the data structure Cust. It is used as a limit for a DO loop.

```
 * Program to read a set of records into a data structure
 ************************************************************
FCustomers IF   E             Disk
D Cust          E Ds                 ExtName(Customers)
D                                    Occurs(10)
D EOF             S               N
D Count           S              2  0
D CustElem        S              2  0
```

The calculation specifications define the *Entry parameter list.

The DO loop reads from the database file and puts the data into data structure Cust, which is passed back as a parameter to the Client program.

The two other parameters indicate the status of the database access:

- EOF is set to ON if the end of file (%EOF) is detected on the READ statement.
- Count contains the number of records being passed back to the client in the data structure Cust.

```
C     *Entry        PList
C                   Parm                    Cust
C                   Parm                    CustElem
C                   Parm                    EOF
C                   Parm                    Count

C                   Eval      Count = 1
C     Count         Occur     Cust
C                   Read(E)   Customers

C                   DoW       Count < CustElem and Not %EOF
C                   Eval      Count = Count + 1
C     Count         Occur     Cust
C                   Read(E)   Customers
C                   EndDo
```

```
     C                   If        %EOF
     C                   Eval      Count = Count - 1
     C                   Eval      EOF = *On
     C                   EndIf

     C                   Return
```

When compiling the server program, be sure not to specify *NEW for Activation Group. If *NEW is specified, any storage allocated by this program is freed when RETURN is executed. This would adversely affect the performance of the application, since the server program would need to be restarted and reopen the database file on each call.

If you use the CRTBNDRPG command to compile this program, specify either `DFTACTGRP(*Yes)` or `DFTACTGRP(*No)` and supply a name for the ACTGRP parameter.

Note: The default activation group name of QILE is acceptable.

If you use the CRTRPGMOD command followed by the CRTPGM command to create this program, be sure to specify a name for the ACTGRP parameter.

One of the benefits of this thin client example is the reusability of the server application by different applications. Even traditional 5250 applications can use the server modules for database access. This approach certainly makes it easier to maintain applications since changes in a server module are reflected in all applications that use it.

8.4.5 Overview diagram

Figure 48 shows a schematic diagram of our example.

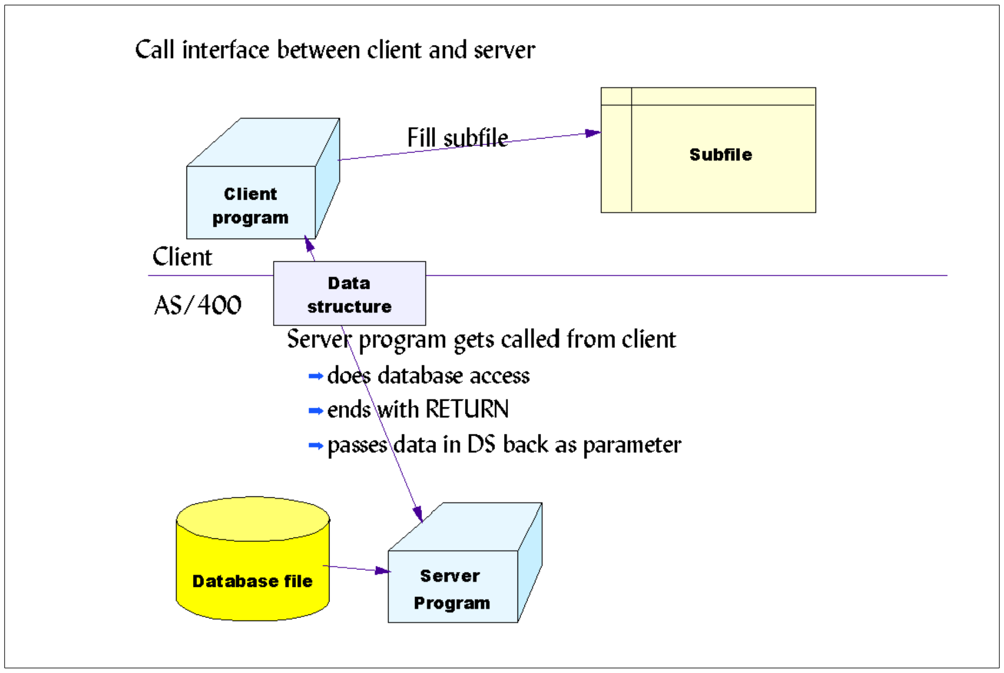


Figure 48. VisualAge for RPG: Example schematic diagram

The client program gets requests from the user interface. It calls a server program that reads records from a database program and passes this data back to the client through parameters. The subfile is filled with the returned data.

8.5 Sample application using data queues

The AS/400 system provides built-in support for data queues to allow applications to communicate with each other asynchronously. This sample application exploits data queues instead of parameter passing to exchange the data from the database with the VARPG client program. This application is based on two data queues on the AS/400 system that are used by the client and the server program. The server program in this example is launched as an independent program on the server using the NOWAIT parameter in the D specs of the client program.

8.5.1 The client application

The user interface is the same as in the previous application. Basically, a subfile is filled with data from the AS/400 server database. The filling of the subfile starts with the create event of the window and continues when the More... push button is pressed or a page end event occurs using the Page Down key. This is essentially the same as in the previous example.

The setup for the data queues is done in the initialization subroutine *INZSR. It calls a program on the AS/400 system to create two data queues in a library on the AS/400 system. To create unique data queues for each client, the last five characters of the IP address are tagged on to the name of the data queues. The characters "I" or "O" at the end of the data queue name provide the unique names for the Input or Output data queues.

The server job receives commands from the "O" data queue. Commands are sent from the client program to the "O" data queue.

After creating the data queues, the server program is started. The two data queue names are passed to it. It then waits on data queue "O" for commands from the client program.

The client program is activated by GUI events and then sends requests to data queue "O". It then waits on data queue "I" until this data queue is filled by the server job.

When the client program gets a termination request, the *TERMSR subroutine is invoked to signal the server program to end, and the two data queues are deleted.

8.5.2 Client sample source

The program is a bit larger because the data queue environment has to be managed in here as well.

8.5.2.1 Data definitions

DLL getHostName is a Windows DLL to get the IP address from the Client to create unique data queue names. Note that we are using a prototyped call to access the Windows DLL from the VARPG program.

```
D* Prototype for DLL
D* This DLL gets the workstation IP address, hostname from WINDOWS TCPIP
D GetHostName     Pr                  ExtProc('getHostName')
D                                     DLL('HOSTNAME.DLL')
D                                     Linkage(*STDCALL)
D                               10A
D                               15A
```

```
D EnthName        S             10A
D EntIPAdd        S             15A
D* Command strings to create and delete data queues
D QCMDEXC         S             10A   Inz('QCMDEXC')
D                                     Linkage(*server)
D Cmd             S            256A
D Cmdlen          S             15P 5 Inz(%Size(Cmd))
D Cmd1            S            256A   INZ('CRTDTAQ DTAQ(QGPL/')
D CmdE            S            256A   INZ('DLTDTAQ DTAQ(QGPL/')
D Cmd2            S              9A   Inz(') MAXLEN(')
D* Prefix for dataq name
D QName1          S              4A   Inz('CUSQ')
D* Variables that contain the 2 dataq names used for one client
D QNameI          S             10A
D QNameO          S             10A
D* Define RCVDTAQ and SNDDTAQ programs as server programs
D QRCVDTAQ        S             10A   Inz('QRCVDTAQ')
D                                     Linkage(*Server)
D QSNDDTAQ        S             10A   Inz('QSNDDTAQ')
D                                     Linkage(*Server)
D* RPGIV server program definition
D DATAQ           S             10A   Inz('DATAQ')
D                                     Linkage(*Server) NoWait
D* Limit for loop
D CustElem        S              2  0  Inz(%Elem(CustDS))
D* Indicator for file end reached
D FileEnd         S               n
D* Parameters for dataq API's
D MsgSz           S              5  0
D NameOfQ         S             10
D NameOfLib       S             10
D Count           S              2  0
D MaxLen          S             10  0  Inz(%Size(CustDs:*All))
D WaitTime        S              5  0

D* Data structure containing data base data
D CustDS        E Ds                  ExtName(Customers) Occurs(10)
D                                     Inz
D* data structure containing process information
D RInfo           Ds
D  EOF                            n
D  NRecords                      2  0
D  Filler                       20
```

8.5.2.2 The initialization subroutine

The data queues are created, and the server RPG program DATAQ is started. The program is invoked with the NOWAIT keyword. This keyword notifies the client program not to wait for it to return. Both programs are working completely asynchronously.

```
 * Initialization subroutine
 * used to setup the server environment
C     *InzSr        BegSr
C* Get client IP addr to build unique dataq names, EntIPAdd contains IP addr
C                   CallP     getHostName(enthname:entipadd)
C* Build Name for 'I' and 'O' Dataq: Add last 5 chars of IP address + I or O
C                   Eval      QNameI= QName1 +
C                              %Subst(EntIPAdd:%Len(%Trim(EntIPAdd))-5:5)
C                              + 'I'
C                   Eval      QNameO= QName1 +
C                              %Subst(EntIPAdd:%Len(%Trim(EntIPAdd))-5: 5)
C                              + 'O'
C* Create data queues
C                   Eval      Cmd = %Trim(%TrimR(Cmd1) + QNameI + Cmd2
C                                 + %Editc(%Size(CustDS:*All):'Z') + ')')
C                   Call(e)   QCMDEXC
C                   Parm                    Cmd
C                   Parm                    CmdLen
C                   Eval      Cmd = *Blank
C                   Eval      Cmd = %Trim (%TrimR(Cmd1) + QNameO + Cmd2
C                                + %Editc(%Size (CustDS:*All) :'Z') + ')')
C                   Call(e)   QCMDEXC
C                   Parm                    Cmd
C                   Parm                    CmdLen

C* Call server program to access data base on server
```

```
C                   Call(e)   DATAQ
C                   Parm                    QNameI
C                   Parm                    QNameO

C* Initialization is done, now do event processing
C                   EndSR
```

8.5.2.3 The action subroutine

A request is sent to data queue "O". Then the client program waits for a response from server program DATAQ on data queue "I". After receiving the data, the subfile is filled in a loop.

```
C     PSBmore       BegAct    PRESS         Win1
C* As long as there is data, get more data
C                   If        Not FileEnd
C*
C* Send request to data queue 'O' to fetch data
C*
C                   Eval      NRecords=10
C                   Call      QSNDDTAQ
C                   Parm                    QNameO
C                   Parm      'QGPL    '    NameOfLib
C                   Parm      23            MsgSz
C                   Parm                    RInfo

C*Wait on dataq 'I' for data
C* Expecting processing data here in DS RInfo
C                   Eval      WaitTime = -1
C                   Eval      MsgSz = 23
C                   Call      QRCVDTAQ
C                   Parm                    QNameI
C                   Parm      'QGPL      '  NAMEOfLiB
C                   Parm                    MsgSz
C                   Parm                    RInfo
C                   Parm                    WaitTime

C* Expecting data from data base here
C                   Eval      MsgSz = %Size(CustDS:*All)
C                   Call      QRCVDTAQ
C                   Parm                    QNameI
C                   Parm      'QGPL      '  NAMEOfLib
C                   Parm                    MsgSz
C                   Parm                    CustDS
C                   Parm                    WaitTime
C* For as many records as the server has read, fill the subfile
C                   Eval      Count = 1

C                   DoW       Count <= NRecords and Not FileEnd
C     Count         Occur     CustDS
C                   Write     Sub1
C                   Eval      Count = Count + 1
C                   EndDo

C* If end of file was signaled, disable the push button
C                   If        EOF
C                   Eval      %Setatr('Win1':'PSBMore':'Enabled') = 0
C
C                   Eval      FileEnd = *On
C                   EndIf
C                   EndIf
C* End of action subroutine
C                   EndAct
```

8.5.2.4 Termination subroutine

A termination request is sent to server program DATAQ, and the two data queues are deleted.

```
C     *TermSr       BegSr
C* Indicate end of program to server program and send data to dataq 'O'
C                   Eval      NRecords = 0
C                   Call(e)   QSNDDTAQ
C                   Parm                    QNameO
C                   Parm      'QGPL    '    NameOfLib
C                   Parm      23            MsgSz
C                   Parm                    RInfo

C* Delete both data queues
C                   Eval      Cmd = *Blank
C                   Eval      Cmd = %Trim(%TrimR(CmdE) + QNameI + ')')
C                   Call(e)   QCMDEXC
C                   Parm                    Cmd
C                   Parm                    CmdLen
C                   Eval      Cmd = *Blank
C                   Eval      Cmd = %Trim(%Trimr(CmdE) + QNameO + ')')
C                   Call(e)   QCMDEXC
C                   Parm                    Cmd
C                   Parm                    CmdLen

C* Application ends
C                   EndSr
```

8.5.3 The server program

After the server program is launched, it enters a loop and waits at data queue "O" until it gets a request from the client program. Two different requests are possible in this example. The program determines which request has been sent to read more data or to terminate.

For a request for more data, it reads 10 more records from the database and then send two items to data queue "I".

The first item contains process information for the number of records that were actually read and whether an end of file situation occurred. The second item contains the multi-occurrence data structure containing the data from the database file. The client program receives these records from data queue "I" and fills the subfile accordingly.

When the server program signals that a termination is requested, the LR indicator is set on, and the DO loop ends. This ends the program. Any other clean up is managed by the client program.

8.5.4 Server sample source

Here is the example source for both the file and data definitions, and in the next section, the main line program.

8.5.4.1 File and data definitions

Here are the definitions of the file and data used in the example program.

```
FCustomers IF   E             Disk
D QRcvDtaQ        PR                  ExtPgm('QRCVDTAQ')
D  QNameO                       10
D  NameOfLib                    10
D  MsgSz                         5  0 Const
D  RInfo                        23
D  WaitTime                      5  0 Const

D QSndDtaQ        PR                  ExtPgm('QSNDDTAQ')
D  QNameI                       10
```

```
D  NameOfLib                    10
D  MsgSz                         5  0 Const
D  RInfo                        23

D* Data structure containing data base data to be passed to client
D CustDS        E Ds                  ExtName(Customers) Occurs(10)
D* Data structure to pass control information between client and server
D RInfo           Ds
D  EOF                           N
D  Count                         2  0
D  Fill                         20

D* Number of occurs in DS for loop limit
D CustElem        S              2  0  Inz(%Elem(CustDS))
D* library name for dataq and data size to be send to dataq and wait time
D NameOfLib       S             10     Inz('QGPL')
D MsgSz           S              5  0
D WaitTime        S              5  0
D* Names of dataq's passed from client
D QNameI          S             10
D QNameO          S             10
```

8.5.4.2 Main line program

Process the DO loop. Wait on data queue "O" until requests arrive. Read more records from the database. Send the data to data queue "I" and wait again for more requests.

```
C*  Beginning of mainline
C     *Entry        Plist
C                   Parm                    QNameI
C                   Parm                    QNameO
C* This DO loop is forever until client program signals that it
C* terminates
C                   DoW       Not *InLR
C* Wait for client program to signal that it needs data
C                   CallP     QRcvDtaQ (QnameO: NameOfLib: 23: RInfo: -1)

C* Read 10 records from database file
C* Count = 0 means client program is terminating
C                   If        Count > 0
C                   Eval      Count = 1
C     Count         Occur     CustDS
C                   Read(e)   Customers
C                   DoW       Count < CustElem and Not %EOF(Customers)
C                   Eval      Count = Count+1
C     Count         Occur     CustDS
C                   Read(e)   Customers
C                   EndDo

C* Determine whether there is more data in file
C                   If        %EOF(Customers)
C                   Eval      Count = Count-1
C                   Eval      EOF = *On
C                   EndIf

C* Send information to the data queue.
C* Send one record with information on how many records are read and
C* whether end of file was reached
C*
C                   CallP     QSndDtaQ (QNameI: NameofLib: 23: RInfo)

C* Send the data in DS from database file to dataq
C     1             Occur     CustDS
C                   CallP     QSndDtaQ (QNameI: NameofLib:
C                                 %Size(CustDS: *All): CustDs)

C* When client program terminates it sends Count 0, then terminate this
C* program as well
C                   Else
C                   Eval      *InLR=*On
C*
C                   EndIf
C                   EndDo
C*
C* End of MAINLINE
```

8.5.5 Overview diagrams

The client and server programs in this example step through a series of states as described in the following sections.

8.5.5.1 Initial state

In the initial state of the application, two data queues are created, and server program DATAQ is started. The server program requests data from data queue "O" and is in an indefinite wait. See Figure 49.

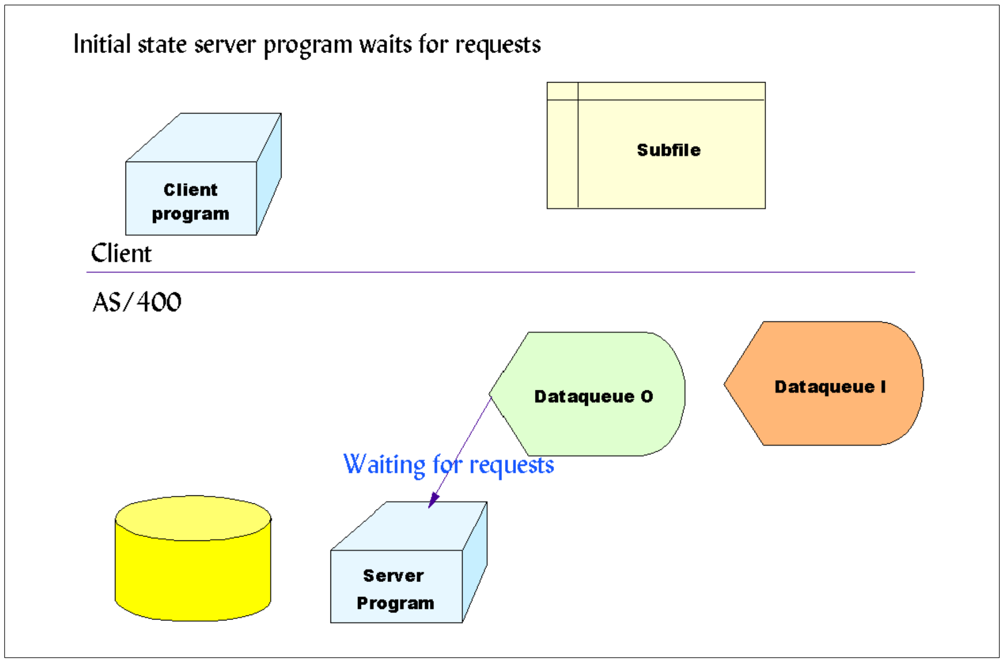


Figure 49. VisualAge for RPG: Initial state of server program

8.5.5.2 Second state

The second state is entered when a GUI event requests more data. The three events that trigger the action subroutine are:

1. Create event from a window.
2. Press an event from the More... push button.
3. Page end an event from a subfile.

See Figure 50.

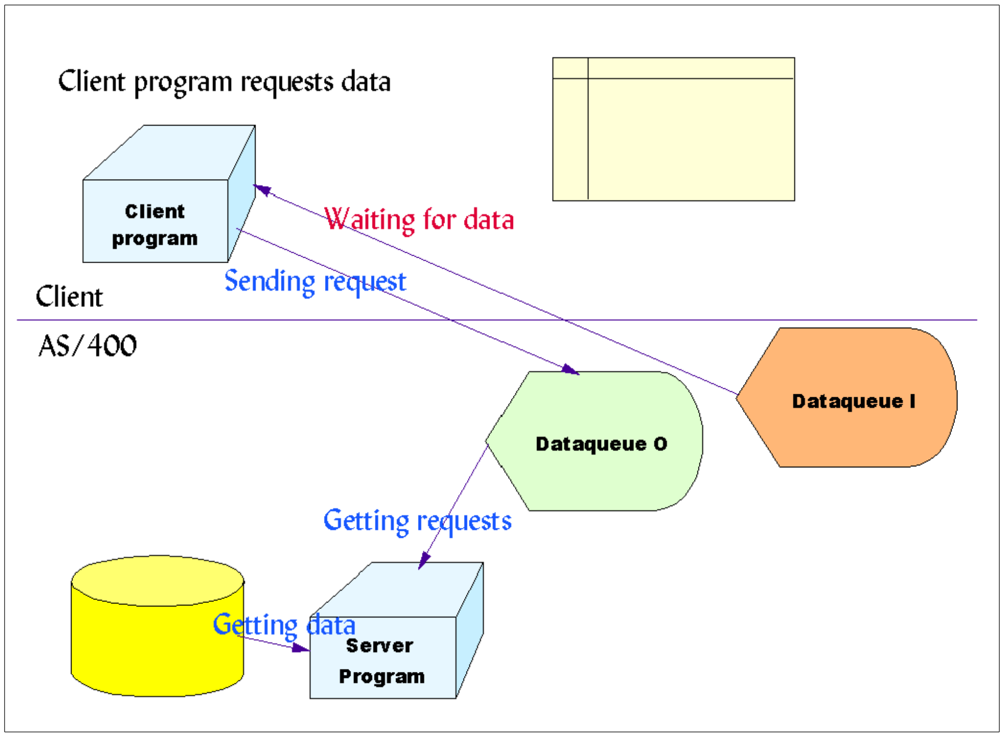


Figure 50. VisualAge for RPG: Client program requests data

The client program then waits on data queue "I" for data. The server program accesses the database file and retrieves the data.

8.5.5.3 Third state

The third state has the server program fill data queue "I". Then, the client program becomes active and puts the data into the subfile. See Figure 51 on page 408.

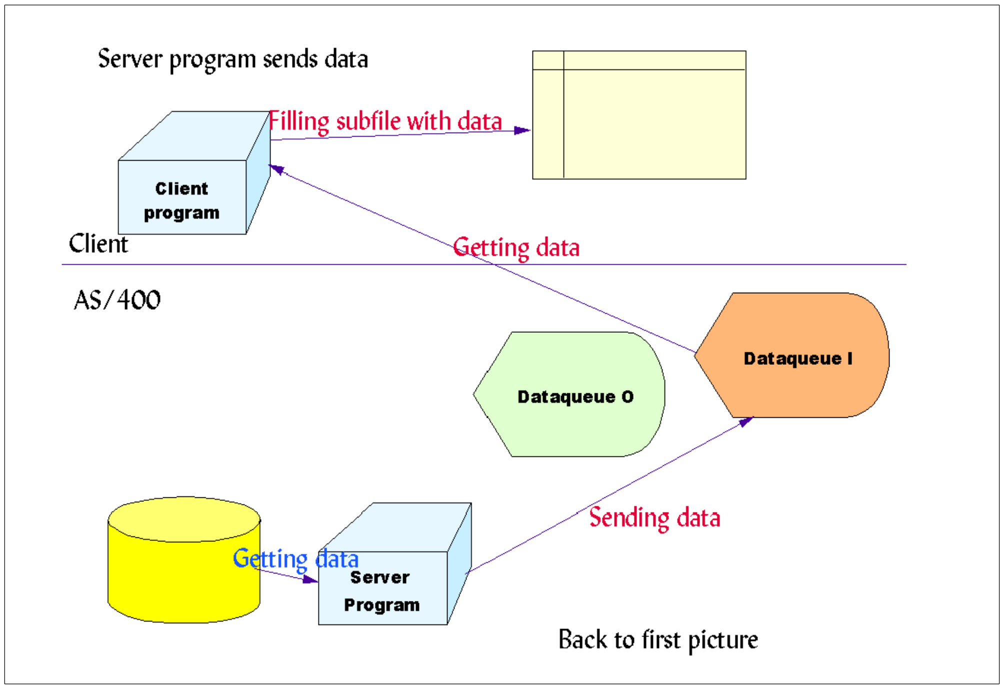


Figure 51. VisualAge for RPG: Server program sends data

After this, the initial state is reached, and the process starts again.

8.6 Variations of these scenarios

In addition to these specific examples, there are variations and combinations of both scenarios possible. The goal here is to minimize the processing on the client and use the power of the server to run these applications. Also easy reuse of modules on the server can be accomplished and even 5250 and GUI applications can share the same server programs in these scenarios.

We know of two variations of the data queue example that are implemented in current production environments by customers of VARPG.

One method uses requests in the form of an SQL statement that is passed to a server program. This server program issues the SQL statement and routes the received data to a data queue. The client program waiting on the data queue uses the data passed back to satisfy the end user request. In this particular application, a single keyed data queue is used instead of multiple data queues (two for each workstation) in our previous example.

Other implementations pass all input data from the user interface to a server program to perform error checking and processing on the server. Any error conditions are passed back to the client. This approach allows high reusability of business logic between 5250 applications and GUI applications and provides for a thin client application.

8.7 Summary

There are several different ways to implement a relatively thin client application model with VARPG. Multiple built-in capabilities of VARPG that provide integration with the AS/400 server platform can be exploited in all of them. In our two examples, three of these built-in capabilities are used:

- **External description of data structures**: Allows you to define externally described data in a data structure easily even if no direct file access is used.
- **Remote call interface**: Provides a simple way to call remote server programs and pass data between them.
- **Reference fields in a GUI designer**: Fields in the subfile of the user interface are defined as reference fields. No additional definition of data base fields is needed.

One of the benefits of Visual Age for RPG is the full support of the RPGIV programming language, so both the server and the client applications can be implemented in the same language.

The tools used (VARPG and CODE/400) to implement the two different applications are similar. To edit server programs, the same editor (CODE/400) is used as for client programs (VARPG).

Copying and pasting code from one application source to the other is easy! You only need to cut and paste using the clipboard. This development environment also allows you to easily change code even if a switch between client program source and server program source is often required during the development phase. CODE/400, the companion product of VARPG, provides the ideal workstation toolset for server program development. When buying VisualAge for RPG, a copy of CODE/400 is included as well, so both tools are readily available to the RPG programmer.

Together with CODE/400, VisualAge for RPG is an ideal development environment to build a client/server application. In this environment, the applications are based on a thin client application model and relies on a robust industry strength server that scales well to any production environment.

8.8 More information VisualAge for RPG

For more information about VisualAge for RPG, visit the IBM VARPG Web site at:
`http://www.software.ibm.com/ad/varpg`

From this Web site, you can (as of this writing) download an evaluation version of the product to try out VARPG for yourself. In addition to the evaluation version, you may also download an interactive tutorial from this Web site. Many other helpful documents and references are also available form this site.

You may also want to check out the book *VisualAge for RPG by Example*, by Bryan Meyers and Jef Sutherland, which is available on the Web at:
`http://www.news400store.com`

Appendix A. Example RPG IV programs on the Web

The RPG programs and AS/400 database libraries used in this redbook are available for you to download from the Internet. These examples were developed using an AS/400 system with OS/400 and the RPG compiler (5769RG1) at V4R4.

The RPGISCOOL SAVF has been saved at a target release of V3R2M0 (TGTRLS(V3R2M0)). This should allow you to restore the library at any release of OS/400 at V3R2M0 or later. Some of the programming examples use functions and features of the compiler and OS/400 system that only compile and run at V4R4 of OS/400. Others may run on a V3R2 or V3R7 system and above with minor modifications.

Important notice

These example programs have not been subjected to any formal testing. They are provided "as is". Use them for reference only. Refer to Appendix D, “Special notices” on page 419, for more information.

A.1 Downloading the files

To use these files, you must download them to your personal computer from the Internet site. A file named README.TXT is included. It contains instructions for restoring the AS/400 library. Go to the site: `http://www.redbooks.ibm.com`

Click on **Additional Materials**, and select the directory **SG245402**. In the **SG245402** directory, click **readme.txt**, and follow the directions.

Appendix B. An introduction to the Integrated File System (IFS)

As Integrated File System (IFS) APIs apply to the integrated file system, this appendix serves to introduce you to this storage directory architecture.

B.1 Introduction

The IFS of OS/400 provides a consistent structure for manipulating all types of information (for example, file types) stored in an AS/400 system. It provides a consistent structure and interface (for users and application programs) to access traditional database files, libraries, folders, documents, and so on, as well as increasingly important information such as images, audio, and video. Support is provided for the following information:

- Stream files, which can contain long continuous strings of data. Examples include the text of a document or the long string of data representing a scanned image.
- CICS files (popular on the IBM S/390 computer family).
- A hierarchical directory structure (similar to that of UNIX and OS/2) that allows access to objects by specifying the path through the directories to the object.
- A common interface that allows users and application programs to access all information, such as stream files, database files, documents, and other objects stored in the AS/400 system.
- Support for the Network File System (NFS) and the Remote File System (RFS) popular in UNIX environments.

The integrated file system enhances the data management capabilities of OS/400 to better support emerging and future forms of information processing (for example, client/server, open systems, and multimedia.) The following benefits are provided by the Integrated File System:

- Fast access to AS/400 data, especially for applications using the PC file server (shared folder) facilities.
- More efficient handling of the increasingly important types of stream data, such as images, audio, and video.
- A file system and directory base for supporting UNIX-based open system standards, such as POSIX and XPG. This file and directory structure also provides a familiar environment for users of UNIX and PC operating systems, such as DOS and OS/2.
- Allows information (such as record-oriented database files, UNIX-based stream files, and file serving) to be handled through separate file systems or managed through a common interface, depending on user needs.
- Allows PC users to take better advantage of the graphical user interface. For example, OS/2 users can use the OS/2 graphical tools to operate on AS/400 stream files and other objects in the same way as they operate on files stored on their PCs.
- Provides continuity of object names and associated object information across national languages. This support ensures that individual characters remain the same when switching from the code page of one language to the code page of another language.

B.2 Integrated File System structure

Figure 52 shows the structure of the OS/400 IFS. From the perspective of structures and rules, the more traditional OS/400 support for accessing database files, documents, and various other object types through libraries can be thought of as a separate file system, which is called QSYS.LIB. Similarly, the new OS/400 support for accessing stream files is a separate file system called QDLS. The IFS consolidates these two file systems (and others) and provides a consistent interface (for users and application programs) to all of the information stored on an AS/400 system.

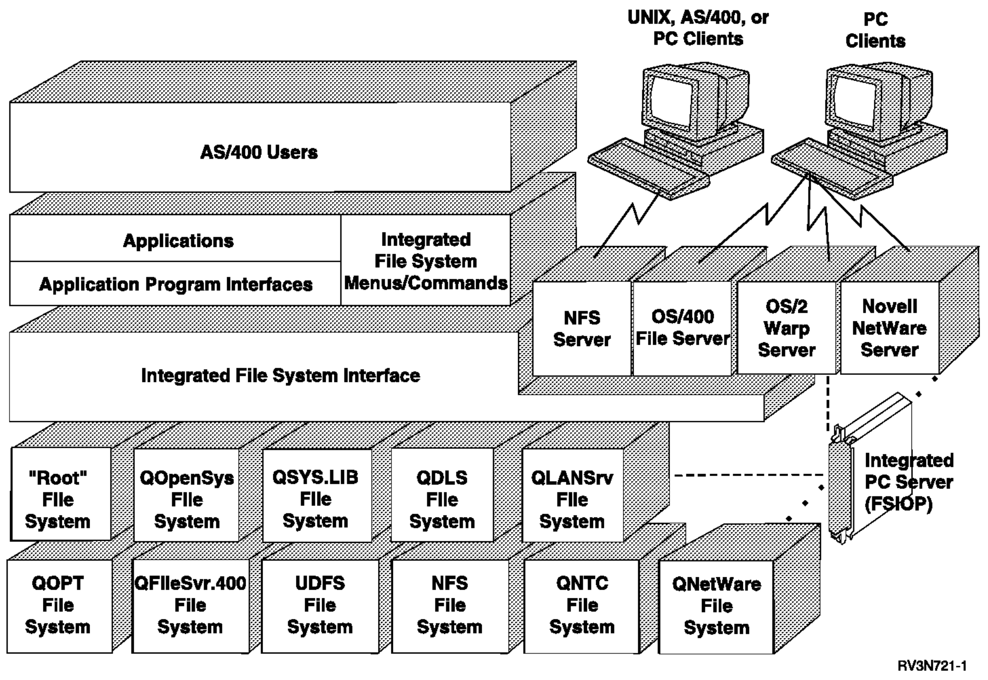


Figure 52. The Integrated File System structure

The following separate file systems are also consolidated under the integrated file system:

- **root:** The *root* file system is designed to take full advantage of the stream file support and hierarchical directory structure of the integrated file system. It has the characteristics of the DOS and OS/2 file systems.
- **QOPENSYS:** The open systems file system is designed to be compatible with UNIX-based open system standards, such as POSIX and XPG. It is stream file oriented and supports case-sensitive object names.
- **QLANSrv**: Formerly known as OS/2 Warp Server, the LAN server file system provides access to the same directories and files as those accessed through the OS/2 Warp Server licensed program. It allows users of the PC file server (shared folders) and AS/400 applications to use the same data as OS/2 Warp Server(5769-XZ1) clients. OS/2 Warp Server runs applications that do not require graphical user interface interaction. OS/2 Warp Server increases save and restore performance, provides printer serving capability, and TCP/IP support including NETBIOS over TCP/IP, and LAN-to-LAN print capability.

The interface provided by the Integrated File System is optimized for input/output of stream data in contrast to the record input/output provided through the traditional OS/400 data management interfaces. A set of common user facilities (commands, menus, and displays) and application program interfaces (APIs) is provided for interacting with all OS/400 file systems (new and old). Triggers, stored procedures, declarative referential integrity, two-phase commit, and long field names are supported under all file forms. Therefore, it improves DB2 for AS/400, as well as CICS/400.

B.2.1 Stream files

A comparison of stream files to database files is useful. A database file is record oriented, which means each item of information (or record) is organized in a predefined format consisting of one or more pieces of information (or fields) that have specific characteristics, such as length and data type. In a traditional database file, all records have the same field structure.

A stream file (Figure 53) contains a continuous stream of data bits with no field or record structure. Documents stored in AS/400 folders are stream files. Other examples of stream files are PC files and the files in UNIX systems.

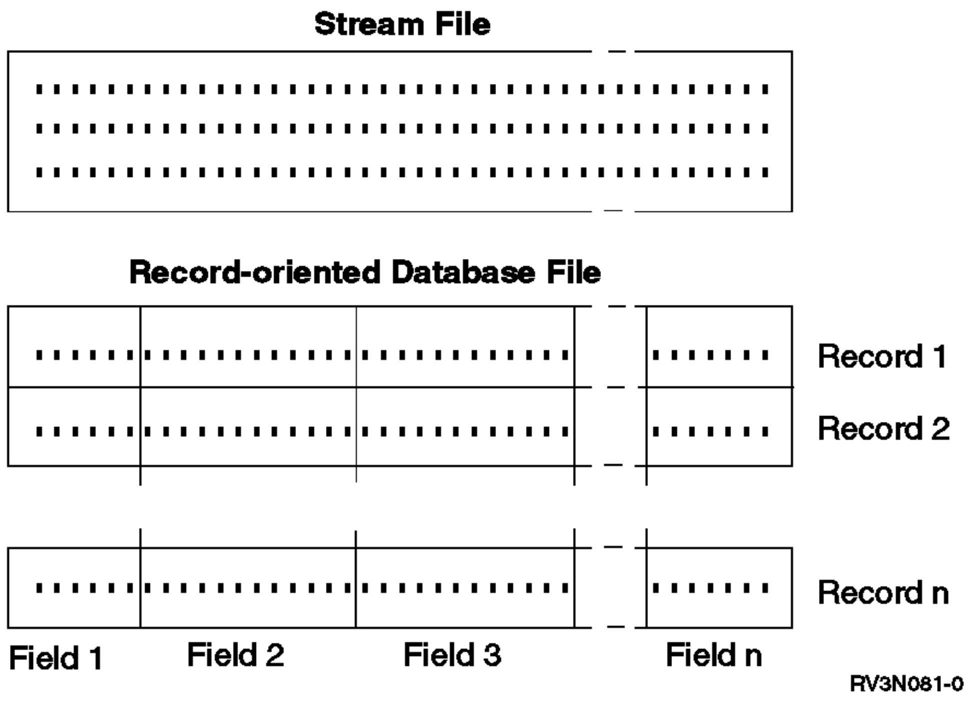


Figure 53. Stream files

The different structures of stream files and record-oriented files lend themselves to different situations. For example, a record-oriented file is well suited for storing customer statistics, such as name, address, and account balance. These predefined fields can be individually accessed and manipulated using the extensive programming facilities of the AS/400 system. A stream file is better suited for storing information, such as a customer's picture, which is composed of a continuous string of bits representing variations in color. Stream files are particularly well suited for storing strings of data, such as the text of a document, images, audio, and video.

B.3 Path name rules using APIs

The following list summarizes the rules you need to keep in mind when specifying path names in the APIs. The term "object" in these rules refers to any directory, file, link, or other object.

- Path names are specified in hierarchical order beginning with the highest level of the directory hierarchy. The name of each component in the path is separated by a slash (/), for example:
  `Dir1/Dir2/Dir3/UsrFile`

  The back slash (\) is not recognized as a separator. It is handled as just another character in a name.
- Object names must be unique within a directory.
- The maximum length of each component of the path name and the maximum length of the path name string can vary for each file system. For more information on the limits of each file system, refer to the AS/400 Information Center found at `http://www.as400.ibm.com/infocenter`

  Once you reach the Information Center site, select **Database and File systems**, and then **Integrated File System**. Go to the section **File System Comparison**, under **File System**.
- A / character at the beginning of a path name means that the path begins at the "root" (/) directory, for example:

  `/Dir1/Dir2/Dir3/UsrFile`
- If the path name does not begin with a / character, the path is assumed to begin at the current directory, for example:

  `MyDir/MyFile`

  Here, MyDir is a subdirectory of the current directory.
- To avoid confusion with AS/400 special values, path names cannot start with a single asterisk (*) character. To specify a path name that begins with any number of characters, use two asterisks (*), for example:

  `'**.file'`

  Note that this only applies to relative path names where there are no other characters before the asterisk (*).
- When operating on objects in the QSYS.LIB file system, the component names must be of the form name.object-type, for example:

  `/QSYS.LIB/PAYROLL.LIB/PAY.FILE`
- Do not use a colon (:) in path names. It has a special meaning within the system.
- Unlike path names in Integrated File System commands, an asterisk (*), a question mark (?), an apostrophe ('), a quotation mark ("), and a tilde () have no special significance. They are handled as another character in a name.

For more information on the IFS, refer to the AS/400 Information Center found at:
`http://www.as400.ibm.com/infocenter`

Once you reach the Information Center site, select **Database and File System**, and then **Integrated File System**. You can also consult the *Integrated File System Introduction V4R3,* SC41-5711.

Appendix C. PTFs for *SRCSTMT and *NODEBUGIO

The following information provides details on the specific PTFs necessary to enable the OPTION *SRCSTMT and *NODEBUGIO features in releases of the RPG IV compiler prior to V4R4, where the support is in the base release. This information is referenced in 1.3.5, “Version 4 Release 4 (V4R4)” on page 11.

You need a compiler PTF for the release where you compile the programs and a runtime PTF for the release where you run your programs. For example, if you compile on V4R2 with TGTRLS(V3R2M0), you need SF47055 on the V4R2 system, and you need SF45189 on the V4R2 system if you want to run your program there. You also need SF45788 on the V3R2 system where you run your program. If you compile and run any programs on your V4R2 system for the *CURRENT release, you also need SF45191.

*Table 98. PTFs for the OPTION *SRCSTMT and *NODEBUGIO features*

Release	Compiler (57xxRG1)	TGTRLS	Runtime (57xxSS1)
V3R2	SF46001	*CURRENT	SF45788
V3R6	SF45749	*CURRENT	SF45430
V3R7	SF46327	*CURRENT	SF46321
	SF47056	*PRV,V3R2,V3R6	
V4R1	SF46327	*CURRENT,*PRV	SF46462
	SF47056	V3R2, V3R6	
V4R2	SF45191	*CURRENT	SF45189
	SF46944	*PRV,V4R1,V3R7	
	SF47055	V3R2	
V4R3	SF45191	*CURRENT	N/A
	SF46944	*PRV,V4R1,V3R7	
	SF47055	V3R2	

Please note that PTF numbers occasionally change over time. To check for the most current PTFs for any product, you can refer to the AS/400 Technical Support Web site for a list of current PTFs. It is located at:

`http://as400service.rochester.ibm.com`

Select the option **Technical Info and Databases**, which leads you to an option to see PTF cover letter information.

Appendix D. Special notices

This publication is intended to help application developers to take full advantage of the RPG IV language and the AS/400 system running V4R4 of OS/400. The information in this publication is not intended as the specification of any programming interfaces that are provided by Operating System/400 or IBM Integrated Language Environment (ILE) RPG for AS/400. See the PUBLICATIONS section of the IBM Programming Announcement for Operating System/400 or IBM Integrated Language Environment (ILE) RPG for AS/400 for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The information about non-IBM ("vendor") products in this manual has been supplied by the vendor and IBM assumes no responsibility for its accuracy or completeness. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

Any pointers in this publication to external Web sites are provided for convenience only and do not in any manner serve as an endorsement of these Web sites.

Any performance data contained in this document was determined in a controlled environment, and therefore, the results that may be obtained in other operating environments may vary significantly. Users of this document should verify the applicable data for their specific environment.

The following document contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples contain the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Reference to PTF numbers that have not been released through the normal distribution process does not imply general availability. The purpose of including these reference numbers is to alert IBM customers to specific information relative to the implementation of the PTF when it becomes available to each customer according to the normal IBM PTF distribution process.

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries:

AS/400	AT
C/400	CICS
CICS/400	CT
DB2	IBM
Integrated Language Environment	Language Environment
MQ	MQSeries
Net.Data	Netfinity
NetView	Network Station
Operating System/400	OS/2
OS/400	RMF
RPG/400	RS/6000
S/390	SP
SQL/400	System/36
System/38	System/390
VisualAge	XT
400	

The following terms are trademarks of other companies:

Tivoli, Manage. Anything. Anywhere.,The Power To Manage., Anything. Anywhere.,TME, NetView, Cross-Site, Tivoli Ready, Tivoli Certified, Planet Tivoli, and Tivoli Enterprise are trademarks or registered trademarks of Tivoli Systems Inc., an IBM company, in the United States, other countries, or both. In Denmark, Tivoli is a trademark licensed from Kjøbenhavns Sommer - Tivoli A/S.

C-bus is a trademark of Corollary, Inc. in the United States and/or other countries.

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

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States and/or other countries.

PC Direct is a trademark of Ziff Communications Company in the United States and/or other countries and is used by IBM Corporation under license.

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation in the United States and/or other countries.

Appendix E. Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

E.1 IBM Redbooks publications

For information on ordering these publications see “How to get IBM Redbooks” on page 427.

- *Building AS/400 Applications with Java*, SG24-2163
- *Complementing AS/400 Storage Management Using Hierarchical Storage Management*, SG24-4450
- *AS/400 Applications: Moving to the 21st Century*, SG24-4790
- *Cool Title About the AS/400 and Internet*, SG24-4815
- *Net.Commerce V3.2 for AS/400: A Case Study for Doing Business in the New Millennium*, SG24-5198

The following publications are available only through the IBM redbooks site at: `http://www.redbooks.ibm.com/`

- *Moving to Integrated Language Environment for RPG IV*, GG24-4358
- *AS/400 Applications: A Fast and Easy Way to Install, Set Up and Work with VRPG and CODE/400 (ADTS CS)*, SG24-4841

Simply type the order number of the publication in the Search field at the top of the window, and click the **Go** button.

E.2 IBM Redbooks collections

Redbooks are also available on the following CD-ROMs. Click the CD-ROMs button at `http://www.redbooks.ibm.com/` for information about all the CD-ROMs offered, updates and formats.

CD-ROM Title	**Collection Kit Number**
System/390 Redbooks Collection	SK2T-2177
Networking and Systems Management Redbooks Collection	SK2T-6022
Transaction Processing and Data Management Redbooks Collection	SK2T-8038
Lotus Redbooks Collection	SK2T-8039
Tivoli Redbooks Collection	SK2T-8044
AS/400 Redbooks Collection	SK2T-2849
Netfinity Hardware and Software Redbooks Collection	SK2T-8046
RS/6000 Redbooks Collection (BkMgr Format)	SK2T-8040
RS/6000 Redbooks Collection (PDF Format)	SK2T-8043
Application Development Redbooks Collection	SK2T-8037
IBM Enterprise Storage and Systems Management Solutions	SK3T-3694

E.3 Other resources

These publications are also relevant as further information sources:

- *MQSeries for AS/400 Application Programming Reference (RPG)*, SC33-1957
- *ILE RPG for AS/400 Programmer's Guide*, SC09-2507
- *ILE RPG for AS/400 Reference*, SC09-2508
- *ILE C Programmer's Guide*, SC09-2712
- *Security - Reference*, SC41-5302
- *OS/400 Sockets Programming V4R4*, SC41-5422
- *ILE Application Development Example V4R1,* SC41-5602
- *ILE Concepts,* SC41-5606
- *ILE C for AS/400 Run-Time Library Reference,* SC41-5607
- *Integrated File System Introduction V4R3,* SC41-5711
- *CL Programming,* SC41-5721
- *AS/400 Advanced Series System API Programming*, SC41-5800
- *AS/400 System API Reference*, SC41-5801
- *DB2 for OS/400 SQL Call Level Interface*, SC41-5806
- *OS/400 UNIX-Type APIs V4R4*, SC41-5875
- Cozzi, Bob. *The Modern RPG Language.* Midrange Computing, 1996 (ISBN 0-9621825-08).
- Cozzi, Bob. *The Modern RPG IV Language*. Midrange Computing, 1999 (ISBN: 1-5834700-26).
- Kernighan, Brian and Plauger, P.J. *The Elements of Programming Style.* McGraw-Hill Book Company, 1998 (ISBN: 0-0703420-75).
- Meyers, Bryan. *RPG/IV Jump Start.* 29th Street Press, 1997 (ISBN: 1-8824196-77)
- Meyers, Bryan. *Starter Kit for the AS/400*. 29th Street Press (Duke Press), 1994 (ISBN 1-882419-09-X).
- Meyers, Bryan. *Control Language Programming for the AS/400*. 29th Street Press, 1997 (ISBN: 1-8824197-66).
- Meyers, Bryan and Sutherland, Jef. *VisualAge for RPG by Example.* 29th Street Press, 1998 (ISBN: 1-8824198-39).
- Popeil, Russ. *RPG Error Handling Technique: Bulletproofing your applications.* 29th Street Press, 1997 (ISBN: 1-8824193-83).

The following publications are available in softcopy only on the Web at:
`http://www.search400.com`

- *DB2 for OS/400 SQL Programming*, SC41-4611
- *DB2 for OS/400 SQL Reference*, SC41-4612

The following publications are available in softcopy only by searching the site at: `http://publib.boulder.ibm.com/pubs/html/as400/online/homeeng1.htm`

- *AS/400e HTTP Server for AS/400 Web Programming Guide,* GC41-5435
- *OS/400 Integrated Language Environment (ILE) CEE APIs V4R4,* SC41-5861
- *System API Reference: OS/400 Message Handling APIs*, SC41-5862
- *System API Reference: OS/400 National Language Support APIs*, SC41-5863
- *System API Reference*: *OS/400 Object APIs*, SC41-5865
- *OS/400 Program and CL Command APIs V4R4,* SC41-5870

E.4 Referenced Web sites

These Web sites are also relevant as further information sources:

- For information on IBM certification programs: `http://www.ibm.com/certify`
- IBM Redbooks home page: `http://www.redbooks.ibm.com`
- IBM Partners in Development Web site: `http://www.as400.ibm.com/developer`
- IBM Information Center: `http://publib.boulder.ibm.com/pubs/html/as400/infocenter.html` or `http://www.as400.ibm.com/infocenter`
- IBM Technical Studio: `http://publib.boulder.ibm.com/pubs/html/as400/techstudio.htm` and `http://www.as400.ibm.com/techstudio`
- IBM online documentation: `http://publib.boulder.ibm.com/html/as400/onlinelib.htm` and `http://as400bks.rochester.ibm.com`
- *IBM Integrated Language Environment (ILE) RPG for AS/400* is located at: `http://www.software.ibm.com/ad/as400/library/ilerpg44.html`
- VisualAge RPG and Code/400 is located at: `http://www.software.ibm.com/ad/varpg`
- A good AS/400 system specific search site can be found at `http://www.search400.com`
- *Experience RPG IV Tutorial* by Rogers, by Masri & Santilli, can be found on the Advice Press Web site at: `http://www.advice.com`
- Robert Cozzi's RPG Web site is located at: `http://www.rpgiv.com`
- *The Modern RPG IV Language*, by Robert Cozzi, can be found at: `http://www.mc-store.com`
- The 400 Group, which has an ILE RPG IV focus area is located at: `http://www.the400group.com`
- Midrange.com home page: `http://www.midrange.com`
- *NEWS/400* has active forums (like a news group) on RPG at: `http://www.news400.com/navbar/Glance-Forums.html`
- The AS/400 user group called COMMON is located on the Web at: `http://www.common.org`
- For European COMMON activities, got to: `http://www.comeur.org/f_events.htm`

- AS/400 Manager's online resource is available at: `http://www.hotlink400.com`
- An eclectic collection of AS/400 code examples from *NEWS/400* Tips and Techniques Community is on the Web at: `http://www.tnt400.com/codepage400.htm`
- *RPG IV Jump Start*, by Bryan Meyers, and *VisualAge for RPG by Example*, by Bryan Meyers and Jef Sutherland, can be found at: `http://www.news400store.com`
- IBM's *AS/400 Magazine*: `http://www.as400magazine.com`
- *Midrange Computing* home page: `http://www.midrangecomputing.com`
- *Midrange Systems*: `http://www.midrangesystems.com`
- Duke Communications home page: `http://www.dukepress.com`
- IBM CGI programming: `http://www.as400.ibm.com/developer/ebiz/cgi`
- For a library of code examples, samples, and tools, got to: `http://www.as400.ibm.com/snippets`
- CGI test cases: `http://hoohoo.ncsa.uiuc.edu/cgi/examples.html`
- IBM AS/400 Custom Technology Center Web site: `http://www.as400.ibm.com/service/welcome_3.htm`
- Easy AS/400 site for RPG programmers: `http://www.easy400.ibm.it`
- For information on MQSeries, go to: `http://www.software.ibm.com/ts/mqseries`
- RPG Developer Network News on RPGIV.com at: `http://www.rpgIV.com/rpgnews/Feb99a/highmath.html` and `http://www.rpgIV.com/rpgnews/Feb99b/timerpg.html`
- Reference topics on CLI programs from Technical Studio: `http://www.as400.ibm.com/tstudio/tech_ref/cli/cli1.htm`
- SQL CLI frequently asked questions: `http://www.as400.ibm.com/db2/clifaq.htm`
- Reference topics on RPG programs from Technical Studio: `http://www.as400.ibm.com/tstudio/db2_400/cli_rpg/rpg_intro.htm`
- AS/400 Technical Support Web site: `http://as400service.rochester.ibm.com`

How to get IBM Redbooks

This section explains how both customers and IBM employees can find out about IBM Redbooks, redpieces, and CD-ROMs. A form for ordering books and CD-ROMs by fax or e-mail is also provided.

- **Redbooks Web Site** http://www.redbooks.ibm.com/

  Search for, view, download, or order hardcopy/CD-ROM Redbooks from the Redbooks Web site. Also read redpieces and download additional materials (code samples or diskette/CD-ROM images) from this Redbooks site.

  Redpieces are Redbooks in progress; not all Redbooks become redpieces and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

- **E-mail Orders**

  Send orders by e-mail including information from the IBM Redbooks fax order form to:

	e-mail address
In United States	usib6fpl@ibmmail.com
Outside North America	Contact information is in the “How to Order” section at this site: http://www.elink.ibmlink.ibm.com/pbl/pbl

- **Telephone Orders**

United States (toll free)	1-800-879-2755
Canada (toll free)	1-800-IBM-4YOU
Outside North America	Country coordinator phone number is in the “How to Order” section at this site: http://www.elink.ibmlink.ibm.com/pbl/pbl

- **Fax Orders**

United States (toll free)	1-800-445-9269
Canada	1-403-267-4455
Outside North America	Fax phone number is in the “How to Order” section at this site: http://www.elink.ibmlink.ibm.com/pbl/pbl

This information was current at the time of publication, but is continually subject to change. The latest information may be found at the Redbooks Web site.

IBM Intranet for Employees

IBM employees may register for information on workshops, residencies, and Redbooks by accessing the IBM Intranet Web site at http://w3.itso.ibm.com/ and clicking the ITSO Mailing List button. Look in the Materials repository for workshops, presentations, papers, and Web pages developed and written by the ITSO technical professionals; click the Additional Materials button. Employees may access MyNews at http://w3.ibm.com/ for redbook, residency, and workshop announcements.

IBM Redbooks fax order form

Please send me the following:

Title	**Order Number**	**Quantity**

First name	Last name	
Company		
Address		
City	Postal code	Country
Telephone number	Telefax number	VAT number

☐ Invoice to customer number

☐ Credit card number

Credit card expiration date	Card issued to	Signature

We accept American Express, Diners, Eurocard, Master Card, and Visa. Payment by credit card not available in all countries. Signature mandatory for credit card payment.

Index

IBM Redbooks evaluation

Who Knew You Could Do That with RPG IV? A Sorcerer's Guide to System Access and More
SG24-5402-00

Your feedback is very important to help us maintain the quality of IBM Redbooks. **Please complete this questionnaire and return it using one of the following methods:**

- Use the online evaluation form found at http://www.redbooks.ibm.com/
- Fax this form to: USA International Access Code + 1 914 432 8264
- Send your comments in an Internet note to redbook@us.ibm.com

Which of the following best describes you?
_ **Customer** _ **Business Partner** _ **Solution Developer** _ **IBM employee**
_ **None of the above**

Please rate your overall satisfaction with this book using the scale:
(1 = very good, 2 = good, 3 = average, 4 = poor, 5 = very poor)

Overall Satisfaction __________

Please answer the following questions:

Was this redbook published in time for your needs? Yes___ No___

If no, please explain:

What other Redbooks would you like to see published?

Comments/Suggestions: (THANK YOU FOR YOUR FEEDBACK!)

SG24-5402-00

Printed in the U.S.A.

Who Knew You Could Do That with RPG IV? A Sorcerer's Guide to System Access and More

SG24-5402-00