AS/400

ILE Application Development Example

Version 4

SC41-5602-00

AS/400

ILE Application Development Example

Version 4

SC41-5602-00

Take Note!

Before using this information and the product it supports, be sure to read the general information under “Notices” on page vii.

First Edition (August 1997)

This edition applies to the licensed program IBM Operating System/400 (Program 5769-SS1), Version 4 Release 1 Modification 0, and to all subsequent releases and modifications until otherwise indicated in new editions.

Make sure that you are using the proper edition for the level of the product.

Order publications through your IBM representative or the IBM branch serving your locality. If you live in the United States, Puerto Rico, or Guam, you can order publications through the IBM Software Manufacturing Solutions at 800+879-2755. Publications are not stocked at the address given below.

IBM welcomes your comments. A form for readers' comments may be provided at the back of this publication. You can also mail your comments to the following address:

IBM Corporation
Attention Department 542
IDCLERK
3605 Highway 52 N
Rochester, MN 55901-7829 USA

or you can fax your comments to:

United States and Canada: 800+937-3430
Other countries: (+1)+507+253-5192

If you have access to Internet, you can send your comments electronically to IDCLERK@RCHVMW2.VNET.IBM.COM; IBMMAIL, to IBMMAIL(USIB56RZ).

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

Contents

Benefits of ILE

ILE Application Example

Discussions

Notices

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 that IBM product, program, or service may be used. Subject to IBM's valid intellectual property or other legally protectable rights, any functionally equivalent product, program, or service may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

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, 500 Columbus Avenue, Thornwood, NY 10594, U.S.A.

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 the software interoperability coordinator. Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Address your questions to:

IBM Corporation
Software Interoperability Coordinator
3605 Highway 52 N
Rochester, MN 55901-7829 USA

This publication could contain technical inaccuracies or typographical errors.

This publication may refer to products that are announced but not currently available in your country. This publication may also refer to products that have not been announced in your country. IBM makes no commitment to make available any unannounced products referred to herein. The final decision to announce any product is based on IBM's business and technical judgment.

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

This publication contains small programs that are furnished by IBM as simple examples to provide an illustration. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. All programs contained herein are provided to you "AS IS". THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE EXPRESSLY DISCLAIMED.

Programming Interface Information

This manual is intended to help you with application programming. It contains general-use programming interfaces which allow you to write programs that use the Integrated Language Environment services in the OS/400 operating system.

Trademarks and Service Marks

The following terms, denoted by an asterisk (*) in this publication, are trademarks of the IBM Corporation in the United States or other countries or both:

AS/400	PDM
C/400	RPG IV
COBOL/400	RPG/400
IBM	SEU
ILE	System/38
Integrated Language Environment	S/38
OS/400	400

Microsoft, Windows, and the Windows 95 logo are trademarks or registered trademarks of Microsoft Corporation.

PC Direct is a trademark of Ziff Communications Company and is used by IBM Corporation under license.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

C-bus is a trademark of Corollary, Inc.

Java and HotJava are trademarks of Sun Microsystems, Inc.

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

About This Book

The purpose of this book is to explain how you can design and develop applications using the Integrated Language Environment (ILE) compilers. The book contains a straightforward application, the steps required to accomplish ILE tasks, and explanations of the code and the steps used. There are also suggestions about how to use the ILE compilers with your current applications. Covered topics include:

- Using ILE versus using the original program model (OPM)
- Gaining the benefits of modular programming in ILE
- Creating modules
- Binding
- Calling programs and procedures
- Running and debugging programs
- Handling exceptions
- Creating service programs
- Using activation groups
- Dealing with performance issues

The design information emphasizes not just how to design applications using ILE, but also why you want to use ILE. Also discussed are the benefits and the trade-offs of your design decisions. To illustrate the design, a very simple and straightforward example was chosen. We acknowledge that the example is unlikely to be implemented in most businesses. However, it was deliberately chosen for its simplicity, and it demonstrates the concepts and possibilities with clarity.

There are four ILE compilers: ILE RPG/400, ILE COBOL/400, ILE C/400, and ILE CL. This book demonstrates how to take advantage of the ILE RPG and CL compilers. To determine exactly how each language works, refer to the programmer's guide for the specific ILE high-level language (HLL).

This book does not describe all of the migration from an existing AS/400 language to an ILE language. That information is contained in the programmer's guide for each ILE language.

Who Should Use This Book

The intended audience for this book is RPG application programmers who are interested in using ILE but who are not very familiar with it. Programmers writing in other languages may also find the examples helpful in understanding the following things:

What ILE is
How it works
How ILE affects programs developed in languages other than RPG

This book is not a revision, an update, or a new edition of *Application Development by Example*. That manual is a tutorial that introduces the novice to programming utilities on the AS/400 system and explains how to design and develop a simple OPM application.

This book, by contrast, focuses on an ILE application. Reading the *Application Development by Example* book before reading this book may be helpful. However, it is not necessary, because programming requirements in ILE are different from those in OPM.

This book assumes that you have already developed RPG applications. It does not assume that you have read the *ILE Concepts* manual, although you are encouraged to do so. Understanding the example contained in this book and the supporting discussions may help you understand the material in the concepts manual better.

How to Use This Book

This book contains lists of coding steps to follow, discussions of those steps, and discussions about the environments in which the coding is done.

At the beginning of Chapters 2 through 7 (Part 2) is a roadmap to the development process discussed in this book. The roadmap highlights the current location and identifies related discussions in Part 3.

How This Book Is Organized

This section explains the purpose of each chapter and appendix in the book.

Part 1, Benefits of ILE

Chapter 1, “Why Should You Use ILE?” contains an overview of ILE and a discussion of the benefits of ILE.

Part 2, ILE Application Example

Chapter 2, “A Sample OPM Application,” describes a simple OPM application (before ILE) using both CL and RPG modules.

Chapter 3, “Converting RPG Source,” shows how OPM RPG code is converted to ILE RPG code. It also explains what is different about an ILE RPG program and what kind of performance it provides.

Chapter 4, “Converting CL Source,” shows how OPM CL code is converted to ILE CL code.

Chapter 5, “Making the RPG Program into a Subprogram,” shows the conversion of an RPG subroutine into a subprogram. Then the program calls the subprogram by using a dynamic call. This is not a step that you would normally take. It is included in order to allow later steps to bind multiple modules together.

Chapter 6, “Creating Modules and Binding by Copy,” demonstrates the most basic form of the ILE binding function by showing the program making a static call to the subprogram. The CL program is then updated to provide more information and is called by a dynamic call and a static call.

Chapter 7, “Making the Subprogram a Service Program,” shows the creation of a simple service program for the date conversion function.

Part 3, Discussions

Chapter 8, “Original Program Model,” discusses some aspects of OPM that you may have taken for granted or may never have thought about. It lays the groundwork for the ILE chapter.

Chapter 9, “Integrated Language Environment (ILE),” contains explanations and discussions of ILE concepts related to the example.

Chapter 10, “Modularity,” discusses the advantages and disadvantages of modular programming.

Chapter 11, “Debugging,” explains some of the features available in the ILE source debugger.

Chapter 12, “Activation Groups,” discusses activation groups, a substructure within a job.

Chapter 13, “Updating ILE Programs,” explains how modules in ILE programs can be replaced.

Chapter 14, “Service Programs,” introduces the concept of service programs. The chapter presents several design approaches and discusses their implications.

Appendixes

Appendix A deals with packaging the Create Program (CRTPGM) and the Create Service Program (CRTSRVPGM) commands.

Appendix B discusses the where-used capability.

Appendix C contains a brief review of the concepts in this book that are related to the example.

Prerequisite and Related Information

For information about other AS/400 publications (except Advanced 36), see either of the following:

- The *Publications Reference* book, SC41-5003, in the AS/400 Softcopy Library.
- The *AS/400 Information Directory*, a unique, multimedia interface to a searchable database that contains descriptions of titles available from IBM or from selected other publishers. The *AS/400 Information Directory* is shipped with the OS/400 operating system at no charge.

Information Available on the World Wide Web

More AS/400 information is available on the World Wide Web. You can access this information from the AS/400 home page, which is at the following uniform resource locator (URL) address:

`http://www.as400.ibm.com`

Select the Information Desk, and you will be able to access a variety of AS/400 information topics from that page.

Benefits of ILE

Chapter 1. Why Should You Use ILE?

You can improve your development environment by using the function provided by the Integrated Language Environment* (ILE*) compilers. Designing ILE applications might take a little time, but the advantages of doing so make the effort worthwhile.

Programs designed specifically for use with the ILE compilers can coexist with your current programs. It is not necessary to change your current programs immediately. To gain some of the advantages that ILE provides, such as system optimization and the source debugger, all you have to do is run the Convert RPG Source (CVTRPGSRC) utility on your existing RPG programs. With careful planning, you can make a gradual transition to ILE. You can design your new applications as ILE modules and convert your current applications to ILE one at a time.

To understand the benefits of converting to ILE, it is necessary to understand what ILE is. This chapter provides a definition of ILE and a brief overview of the benefits. For more detailed information, see the *ILE Concepts* manual.

Overview of the Integrated Language Environment

The Integrated Language Environment (ILE) is a new set of tools and associated system support. ILE is designed to enhance program development and maintenance on the AS/400 system. In addition to enhancements to the OS/400* operating system, ILE includes a new family of compilers:

ILE RPG/400*
ILE COBOL/400*
ILE C/400*
ILE CL

ILE Benefits

ILE offers numerous benefits not found on previous releases of the AS/400 system. These benefits include:

- Better call performance
- Modularity
- Multiple-language integration
- Enhancements to the ILE compilers
- Reusable components
- Control over application run-time environment
- Code optimization
- Tool availability
- Foundation for the future

In addition, ILE offers common run-time routines used by the ILE-conforming languages. Many of the application program interfaces (APIs) are also provided as bindable (service) programs. This allows your applications to use APIs and to get faster ILE call performance. These off-the-shelf components provide such services as:

Date manipulation
Message handling

Math routines

For details about the APIs supplied with ILE, see the *System API Programming* book.

Better Call Performance

An ILE compiler does not produce a program that can be run. Instead, it produces a **module** object (*MODULE) that can be combined, or **bound**, with other modules to form a single runnable unit, or program. ILE programs are called just as you call programs in your current applications.

A benefit of this binding process is that it helps to reduce the overhead associated with calling programs by reducing the number of external calls. Before ILE, only dynamic (or external) program calls were available to the application programmer. With ILE, two kinds of calls are available:

Dynamic (or external program) calls
Static (or bound) calls

The performance of dynamic calls in ILE programs is fairly close to existing call performance. However, bound calls offer better performance than dynamic calls. Thus, the binding capability and the improved call performance that results may encourage you to develop your applications with a more modular design.

Modularity

A more modular approach to programming provides numerous benefits to you, including:

- Faster compilation because the units of code to compile are smaller (especially in recompiling during development).
- Better programmer work load distribution.
- Opportunities to both purchase and sell individual modules of code.
- Increased ability to reuse a piece of code in a variety of applications. Modules written for a specific function can be bound into several program objects.
- Simplified maintenance. Maintenance may be required in only a single module.

Multiple-Language Integration

With your current application, you can mix different language programs, such as RPG, COBOL, and C. However, to access code written in another language, your current application must perform a dynamic call to a separate program. The performance cost of the dynamic call to a program and the inconsistencies between language behaviors sometimes complicate the mixing of languages.

With ILE, modules written in any ILE language can be bound to modules written in the same or any other ILE language. For example, a module of code written in ILE C/400 (perhaps a floating-point calculation) can be bound with modules written in ILE RPG/400, ILE COBOL/400, ILE C/400, or ILE CL.

This produces a better performing, and more easily managed, application. In addition, you can acquire modules written in a variety of languages, without needing to produce the code yourself. The APIs that IBM provides for ILE are just the beginning. Vendors have more freedom to sell (and application programmers to buy) libraries of routines for any commonly used function, such as tax calculations. They can be written in any language and can be bound for better performance.

Enhancements to the ILE Compilers

The ILE compilers have some significant new function included as part of the language. This is particularly true for ILE RPG/400, which is based on the RPG IV language definition. Many long-standing requests from RPG programmers have been addressed in the ILE RPG/400 compiler, including the following:

- 10-character field names
- Free-form logical and math expressions
- Date and time data types and operations
- External data items (data export)
- Uppercase and lowercase source
- File-level field prefix support
- Pointers

For many programmers, the primary motivation for moving to ILE is to get access to the function that ILE language support provides.

Control over Application Run-Time Environment

ILE allows you to use better control over your application and the resources it uses. You can specify that a given ILE program run in a particular area within a job. This area within a job is called an **activation group**. You can assign a name to the activation group within the job. Then, ILE programs and service programs can be created to use the named activation group. Thus, you can use activation groups to set up logical boundaries within the job to separate the applications.

Within these boundaries, an activation group has exclusive use of the resource, such as open data paths for the files used in the application.

Using activation groups to isolate applications can also make it easier to end an application in a job. It aids in cleaning up its resources (such as open files and active programs) without disturbing resources associated with other applications active in the job. RPG programmers might think of this technique as a kind of application-level LR indicator. For example, it is a way to end an entire application rather than ending one program at a time.

Code Optimization

The new ILE compilers and the associated OS/400 translator have more advanced optimization techniques built into them. In some cases, these new levels of optimization may lead to improved performance of existing code. At compilation time, the programmer can select the desired level of optimization for each piece of code.

Tool Availability

The majority of tools for developers in the computer industry today are written in the C language. With ILE binding capability and improved optimization, these C language applications run faster. In addition, they perform better than they did with the previous C/400 compiler.

Therefore, we anticipate that many tool vendors will begin to add their tools to the AS/400 to attract a new marketplace for their products. Making use of the C language offers you a greater choice of:

CASE tools
Fourth-generation languages (4GLs)
Editors

Debuggers

Foundation for the Future

In addition to the increased opportunity to optimize your applications with the current ILE compilers, you can look forward to even more significant enhancements. The move toward object-oriented programming languages and visual programming tools increases the need for the capabilities provided by ILE.

Applications constructed of large numbers of small, modularized, reusable components that efficiently transfer control among themselves offer you maximum flexibility. You can use them multiple ways in multiple applications.

Summary of ILE

This book provides an example of how you can bring an existing application into ILE and gain some of the benefits immediately, such as the following:

New language function
Better code optimization
New debugger

Gaining full advantage of ILE benefits may inspire you to do some work on your current applications, such as restructuring the code into reusable modules. The remainder of this book gets you started on the road to enjoying some ILE benefits. Your best opportunity to fully exploit ILE is with modular design and application control techniques in future application development or major renovation projects.

The following chart summarizes the similarities and differences between the original program model (OPM) and ILE that are illustrated in this book's examples.

Table 1-1. Similarities and Differences between OPM and ILE

	OPM	**ILE**
Program object types	Program	Program, service program
Results of compilation	Runnable program	Nonrunnable module object
Steps in program-creation process	Compile, run	Compile, bind, run
Run unit	Simulated for each language	Activation group
Types of program calls	Dynamic	Dynamic, static
Focus	Single language	Mixed language
Error handling	Language-specific	Common, language-specific
Debugger	OPM	Source-level, OPM

ILE Application Example

Chapter 2. A Sample OPM Application

Roadmap

Steps in example (current step is highlighted):

- **A Sample OPM Application**
- Converting RPG Source
- Converting CL Source
- Making the RPG Program into a Subprogram
- Creating Modules and Binding by Copy
- Making the Subprogram a Service Program

Related discussions:

- Original Program Model
- Integrated Language Environment

This chapter introduces a simple OPM application (a date conversion) and discusses how it is created on AS/400. This same application is used in later chapters, but is changed to demonstrate the new ILE functions.

The application is simple because its purpose is to demonstrate meaningful function with ILE features, not to provide a sophisticated application design. Later in this book, some alternatives are discussed, including the new RPG IV feature of date conversion.

A Simple OPM Application

Assume that you are an RPG programmer and that you are asked to provide a listing of all the objects found in a particular library. This can be done with a system command, Display Library (DSPLIB). However, to demonstrate ILE principles, assume that you develop an application instead.

The program should work for any specified library. The program obtains a list of the objects in the library and creates a spooled printer file. The listing includes one line for each object. Each line contains the following information:

Object name
Object type
Object attribute
Date last used in a format of YYMMDD

If you are given this task, you must first design an approach to this application. There are two ways to gain access to a list of objects in a library:

- Use an application program interface (API)
- Use the Display Object Description (DSPOBJD) command and create an output file that can be read

Designing a CL Program in OPM

Assume that you decide to use the DSPOBJD command. Because CL programs do not provide printer output, you need an RPG program in order to get a printed listing. Thus, a simple approach is to use the following two programs:

1. A CL program to create the outfile
2. An RPG program to read the outfile, format the data, and print the output (one line per object)

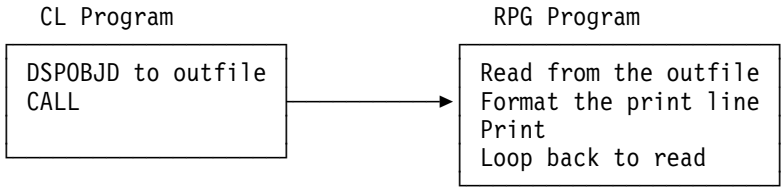

Figure 2-1. An Existing OPM Application

You decide to name the programs PRTPGMC (print program CL) and PRTPGMR (print program RPG). Then you pass to the CL program a parameter that specifies the library to be analyzed. For example, if you want to print a listing of the objects in library QGPL, you would enter:

```
CALL    PGM(PRTPGMC) PARM(QGPL)
```

Normally you enter source statements by using the source entry utility (SEU) menu option of the programming development manager (PDM). The steps described below allow you either to perform them on the system or just to read them in this book.

1. Assume that you create a library named PRTLIB for this application.

   ```
   CRTLIB  LIB(PRTLIB) TEXT('Sample ILE Application')
   ```

2. To make PRTLIB the current library, you enter the following command:

   ```
   CHGCURLIB  CURLIB(PRTLIB)
   ```

   All of the examples in this book assume that PRTLIB is the current library.

3. You create a source file named SOURCE in library PRTLIB. For RPG IV, a record length of 112 is recommended, so you change the default length of 92 to 112.

   ```
   CRTSRCPF  FILE(SOURCE) RCDLEN(112) TEXT('Sample ILE Application')
   ```

4. You use PDM to work with the source files:

   ```
   WRKMBRPDM   FILE(PRTLIB/SOURCE)
   ```

5. To create the CL source file, you press function key F6=Create.

6. Assign the member for the CL program (PRTPGMC) a source type of CLP and a text description of `Print program - OPM`. The source is entered as shown in Figure 2-2 on page 2-3.

   Note: The line numbers (for example, 00.10) at the left margin of the example are entered automatically for you.

```
00.10/* PRTPGMC - Print program - OPM                          */
00.20/*                                                       */
00.30             PGM        PARM(&LIB)
00.40             DCL        &LIB *CHAR LEN(10)
00.50             DLTF       FILE(QTEMP/DSPOBJDP)
00.60             MONMSG     MSGID(CPF2105) /* Ignore if not found */
00.70             DSPOBJD    OBJ(&LIB/*ALL) OBJTYPE(*ALL) +
00.80                          OUTPUT(*OUTFILE) OUTFILE(QTEMP/DSPOBJDP)
00.90                        /* Override the name of the model   */
01.00                        /*   file in QSYS used by the       */
01.10                        /*   DSPOBJD command to the outfile */
01.20                        /*   created in QTEMP               */
01.30             OVRDBF     FILE(QADSPOBJ) TOFILE(QTEMP/DSPOBJDP) +
01.40                          SECURE(*YES)
01.50             CALL       PGM(PRTPGMR) PARM(&LIB)
01.60             ENDPGM
```

Figure 2-2. Source for OPM CL Program PRTPGMC

Here is a brief explanation of the source coding. The numbers at the left are line numbers.

00.40 The &LIB parameter is passed into the program and identified on the PGM statement. This parameter is used later in the program.

00.50 The program uses the Delete File (DLTF) command to delete the file DSPOBJDP in library QTEMP. The program wants to ensure that no file of this name exists in QTEMP before it begins to create the outfile.

00.60 If the file does not exist, an escape message (CPF2105) is issued by the system. The escape message is monitored for, and the condition is ignored.

00.70 The Display Object Description (DSPOBJD) command is used to create an outfile in QTEMP. All object types are requested. When the command completes, the DSPOBJDP file in QTEMP contains one record for each object found in the library.

01.30 The name of the file in QSYS that is used by DSPOBJD is QADSPOBJ. This name is used in the RPG program in order to compile against a file that has the correct format. An Override Database File (OVRDBF) command is needed to override the file used in the RPG program, using instead the file that was created in QTEMP.

7. After the source is entered, you create the program by using option 14 from the PDM menu and place the object program in the PRTLIB library. When you use the Create option, PDM generates the create command and submits it to batch. PDM determines which create command to use based on the type of source that was entered. In this case the source type is CLP, so PDM generates the Create CL Program (CRTCLPGM) command:

```
CRTCLPGM   PGM(PRTLIB/PRTPGMC) SRCFILE(PRTLIB/SOURCE)
```

Designing an RPG Program in OPM

The source for the RPG program (PRTPGMR) is entered with a source type of RPG and a text description of `Print program - OPM`. The source for program PRTPGMR is shown in Figure 2-3.

```
00.10      * PRTPGMR - Print program - OPM
00.20      *
00.30      * QADSPOBJ is the outfile from DSPOBJD - Override occurs in CL
00.40     FQADSPOBJIF  E                    DISK
00.50     FQPRINT  O   F     132     OF     PRINTER
00.60      ****************************************************************
00.70      * Parameter list
00.80     C           *ENTRY    PLIST                           Parm list
00.90     C                     PARM           LIB    10        Library
01.00     C                     EXCPTHDG                        Prt heading
01.10      ****************************************************************
01.20      * Read a record
01.30      * QLIDOBJD is the format name of the QADSPOBJ file
01.40     C                     READ QLIDOBJD                20 Read
01.50      * Continue reading until EOF
01.60     C           *IN20     DOWEQ'0'                        Not EOF
01.70      ****************************************************************
01.80      * Use a subroutine to convert the date from MMDDYY to YYMMDD
01.90     C                     MOVE ODUDAT    MMDDYY  6        MMDDYY fmt
02.00     C                     EXSR CVTDAT                     Convert date
02.10     C                     MOVE YYMMDD    LSTUSD  60       Last used dt
02.20     C                     EXCPTDETAIL                     Print detail
02.30     C   OF                EXCPTHDG                        Prt heading
02.40     C                     READ QLIDOBJD                20 Read
02.50     C                     ENDDO                           Loop Back
02.60      * End the program
02.70     C                     SETON                     LR    Set LR
02.80      ****************************************************************
02.90     C           CVTDAT    BEGSR
03.00      * Convert date from MMDDYY to YYMMDD format
03.10     C                     MOVE MMDDYY    WORK2   2        Move YY
03.20     C                     MOVELWORK2     YYMMDD  6        Move YY
03.30     C                     MOVELMMDDYY    WORK4   4        Move MMDD
03.40     C                     MOVE WORK4     YYMMDD           Move MMDD
03.50     C                     ENDSR
03.60      ****************************************************************
03.70     OQPRINT  E  206           HDG
03.80     O                                    25 'Objects '
03.90     O                                       'in Library - '
04.00     O                         LIB
04.10     O        E  2             HDG
04.20     O                                     6 'Object'
04.30     O                                    18 'Obj type'
04.40     O                                    30 'Attribute'
04.50     O                                    42 'Last used'
04.60     O        E  1             DETAIL
04.70     O                         ODOBNM    10
04.80     O                         ODOBTP    19
04.90     O                         ODOBAT    33
05.00     O                         LSTUSDY   41
```

Figure 2-3. Source for OPM RPG Program PRTPGMR

The following is a brief explanation of the source coding:

00.40 The externally described database file is named QADSPOBJ. This is the name of the file provided by the system that is used when an outfile is created by the DSPOBJD command. The system essentially copies the format of the model file to the outfile named on DSPOBJD. The RPG program describes the file in QSYS because it wants to use the format that exists in the file. An OVRDBF command in the CL program causes the DSPOBJDP file in QTEMP to be read instead of the file that is compiled into the program. Because both files use the same format, the compiled RPG program knows what the layout of the format is.

00.50 The QPRINT file is described as a fixed-format file. Since no OVRPRTF command was specified prior to the open operation of QPRINT, the system provides access to the first QPRINT file found on the library list. This is probably the QPRINT file supplied by IBM in the QGPL library. This file causes spooled printed output to occur.

00.80 The PLIST and PARM operation codes provide the means to pass in the variable library name from the CL program. This library name is printed on the heading line.

01.00 The EXCPT statement prints the heading line as output.

01.40 The program reads a record using the QLIDOBJD format. This is the name of the format used in the QADSPOBJD outfile.

01.60 If end of file is detected, the program sets the LR indicator on and returns.

01.80 A subroutine is used to convert the date-last-used field (which exists in the outfile in a MMDDYY format) to a YYMMDD format. There are many solutions to the problem of reformatting a date field. In this example, the subroutine method is used because in a later chapter we change from an RPG subroutine to a subprogram.

01.90 The program moves the ODUDAT field (date last used) to a common field (MMDDYY) and then calls the subroutine. This type of coding creates a general-purpose subroutine that can convert any MMDDYY date format. When the subroutine returns, the YYMMDD field is moved to the field named LSTUSD, which is a decimal field, so that editing can occur.

02.20 A detail line is printed.

02.30 If printer overflow occurs, an EXCPT operation is used to print the heading on the next page.

02.40 The program loops back to read another record.

03.10 The subroutine converts the date to the correct format by using MOVE and MOVEL operations. There are many other ways to alter the date format, but this method allows the same code to move to a subprogram with ease in a later chapter.

03.70 The heading output includes the library name that was passed into the program.

04.60 The detail line prints several fields.

After the source is entered, you create the program by using option 14 (Create) from the PDM menu and place the object program in the PRTLIB library. When

you use the Create option, PDM generates the create command and submits it to batch. The command to create an OPM RPG program is:

```
CRTRPGPGM   PGM(PRTLIB/PRTPGMR) SRCFILE(PRTLIB/SOURCE)
```

The PDM Work Display in OPM

After you enter both source members, they are listed in the *Member* column of the Work with Members Using PDM display.

```
                          Work with Members Using PDM

File  . . . . . .   SOURCE
  Library . . . .     PRTLIB              Position to  . . . .

Type options, press Enter.
  2=Edit         3=Copy       4=Delete       5=Display      6=Print
  7=Rename       8=Display description       9=Save         13=Change text

Opt  Member      Type        Text
 _   PRTPGMC     CLP         Print programs in a library
 _   PRTPGMR     RPG         Print programs in a library - Called



Parameters or command
===>
F3=Exit          F4=Prompt             F5=Refresh            F6=Create
F9=Retrieve      F10=Command entry     F23=More options      F24=More keys
                                         (C) COPYRIGHT IBM CORP.
```

Figure 2-4. Example of Work with Members Using PDM Display

The *Type* column is very important. It determines the syntax checking that is used by SEU. The syntax checking for each type of source is very different. When you create an object (option 14), PDM uses the type to determine the correct create command to use as described earlier.

Assume that you add library PRTLIB to your library list and call the CL program as follows:

```
ADDLIBL   PRTLIB
CALL      PGM(PRTPGMC) PARM(PRTLIB)
```

The spooled output shows three objects, two of type *PGM and one of type *FILE.

```
      Programs and Modules in Library - PRTLIB

Object    Obj type   Attribute   Last used

PRTPGMC    *PGM        CLP       93/12/10
PRTPGMR    *PGM        RPG        0/00/00
SOURCE     *FILE       PF        93/12/10
```

Figure 2-5. Spooled Output from CL Program Call

Chapter 3. Converting RPG Source

Roadmap

Steps in example (current step is highlighted):

- A Sample OPM Application
- **Converting RPG Source**
- Converting CL Source
- Making the RPG Program into a Subprogram
- Creating Modules and Binding by Copy
- Making the Subprogram a Service Program

In this chapter, we show how to convert OPM RPG source code to ILE RPG source code. If you are coding these examples on a system as you read through this book, follow these steps. Begin by copying the source you used previously to make new source members that you will change into ILE versions of the program.

1. Start PDM:

   ```
   WRKMBRPDM   FILE(PRTLIB/SOURCE)
   ```

2. Use the Copy option to copy the CL program source (PRTPGMC). Name the copy PRTPGMC2.
3. End SEU and use PDM Option 14 to create the program. At this point, the program is still an OPM version.
4. Change the CALL command to:

   ```
   CALL        PGM(PRTPGMR2) PARM(&LIB)
   ```

RPG IV

ILE RPG/400 is a new RPG compiler that is based on the RPG IV language. In addition to providing the proper internals for ILE support, the new compiler supports many new functions, such as:

10-character field names
New D specifications to provide a simpler and more powerful data description
Expanded calculation specifications
A new EVAL operation to allow expressions
Pointer-based processing

The most significant difference is the format of the specifications. You have to convert the OPM RPG (RPG III) source to the new ILE RPG (RPG IV) source format before you can create an ILE program.

There is also a change in the GENOPT parameter on the ILE CL commands Create RPG Module (CRTRPGMOD) and Create Bound RPG Program (CRTRPGPGM). The GENOPT(*LIST) function that listed the intermediate form of the program (the MI instructions) is no longer available. The intermediate form is now considered proprietary.

A conversion function is available that converts the OPM form of RPG to the new ILE form, RPG IV. It is the Convert RPG Source (CVTRPGSRC) command. The

main area of possible conversion problems is in the use of /COPY members. For more information, see the *ILE RPG/400 Programmer's Guide*.

When you run the conversion process, you convert from source type RPG to source type RPGLE. The new RPGLE type is used by PDM to determine the correct syntax checking for the new RPG create command.

Convert RPG Source (CVTRPGSRC) Command

The Convert RPG Source (CVTRPGSRC) command allows you to copy the new specifications to one of the following:

- A different member in the same file
- The same or a different member in a different file

Assume that you want to use CVTRPGSRC to convert to a different member in the same file. Perform the following steps:

1. Convert the OPM RPG source to ILE RPG:

   ```
   CVTRPGSRC  FROMFILE(PRTLIB/SOURCE) FROMMBR(PRTPGMR)
                TOFILE(PRTLIB/SOURCE) TOMBR(PRTPGMR2)
   ```

2. Start PDM if you are not already there.

   ```
   WRKMBRPDM FILE(PRTLIB/SOURCE)
   ```

3. If you are already on the PDM work display and do not see the PRTPGMR2 member, use F5 to refresh the list of members. Note that the source type for the PRTPGMR2 is RPGLE.

4. Change the text of the member to `Print program - ILE`.

5. Use SEU to edit PRTPGMR2. Change the comment on line 00.10 as follows:

   ```
   * PRTPGMR2 - Print program - ILE
   ```

   The source for the converted program PRTPGMR2 should look like Figure 3-1 on page 3-3.

```
00.10      * PRTPGMR2 - Print program - ILE
00.20      *
00.30      * QADSPOBJ is the output from DSPOBJD - Override occurs in CL
00.40     FQADSPOBJ  IF   E             DISK
00.50     FQPRINT    O    F  132        PRINTER OFLIND(*INOF)
00.60      ****************************************************************
00.70      * Parameter list
00.80     C     *ENTRY        PLIST                                          Parm list
00.90     C                   PARM                    LIB             10     Library
01.00     C                   EXCEPT    HDG                                  Prt heading
01.10      ****************************************************************
01.20      * Read a record
01.30      * QLIDOBJD is the format name of the QADSPOBJ file
01.40     C                   READ      QLIDOBJD                       20     Read
01.50      * Continue reading until EOF
01.60     C     *IN20         DOWEQ     '0'                                   Not EOF
01.70      ****************************************************************
01.80      * Use a subroutine to convert the date from MMDDYY to YYMMDD
01.90     C                   MOVE      ODUDAT        MMDDYY            6     MMDDYY fmt
02.00     C                   EXSR      CVTDAT                                Convert date
02.10     C                   MOVE      YYMMDD        LSTUSD            6 0   Last used dt
02.20     C                   EXCEPT    DETAIL                                Print detail
02.30     C   OF              EXCEPT    HDG                                   Prt heading
02.40     C                   READ      QLIDOBJD                       20     Read
02.50     C                   ENDDO                                           Loop Back
02.60      * End the program
02.70     C                   SETON                                      LR   Set LR
02.80      ****************************************************************
02.90     C     CVTDAT        BEGSR
03.00      * Convert date from MMDDYY to YYMMDD format
03.10     C                   MOVE      MMDDYY        WORK2             2     Move YY
03.20     C                   MOVEL     WORK2         YYMMDD            6     Move YY
03.30     C                   MOVEL     MMDDYY        WORK4             4     Move MMDD
03.40     C                   MOVE      WORK4         YYMMDD                  Move MMDD
03.50     C                   ENDSR
03.60      ****************************************************************
03.70     OQPRINT    E            HDG            2 06
03.80     O                                          25 'Objects '
03.90     O                                             'in Library - '
04.00     O                       LIB
04.10     O          E            HDG            2
04.20     O                                           6 'Object'
04.30     O                                          18 'Obj type'
04.40     O                                          30 'Attribute'
04.50     O                                          42 'Last used'
04.60     O          E            DETAIL         1
04.70     O                       ODOBNM             10
04.80     O                       ODOBTP             19
04.90     O                       ODOBAT             33
05.00     O                       LSTUSD        Y    41
```

Figure 3-1. Source for ILE RPG Program PRTPGMR2

6. End SEU and use PDM option 14 to create a program. Because the type is RPGLE, PDM uses the Create Bound RPG Program (CRTBNDRPG) command instead of the Create RPG Program (CRTRPGPGM) command.

Create Bound RPG Program (CRTBNDRPG) Command

The CRTBNDRPG command is submitted to batch as:

```
CRTBNDRPG PGM(PRTLIB/PRTPGMR2) SRCFILE(PRTLIB/SOURCE)
          SRCMBR(PRTPGMR2)
```

When you use CRTBNDRPG, the system first creates a temporary module from your source and then creates a program. There is a separate command, Create RPG Module (CRTRPGMOD), that creates a permanent module object.

The CRTBNDRPG command differs from the Create RPG Program (CRTRPGPGM) command because CRTBNDRPG creates an ILE program consisting of a single module. A module is the real output of the ILE RPG compiler. CRTBNDRPG goes an extra step in creating a program. CRTBNDRPG also has a BNDDIR parameter that can be used to bind multiple modules into a single ILE program object.

The output of the CRTBNDRPG and the CRTRPGPGM commands is the same in that they both create a program object that can be dynamically called. The output is different in that CRTBNDRPG creates an ILE program whereas CRTRPGPGM creates an OPM program. The two programs do not share the same structure.

Modules

A module is not a program, but it can either be made into a program or combined with other modules and made into a program. The important things to know about modules are the following:

- Modules are separate object types.
- Modules are not runnable. Modules, no matter where they are located (even if not bound into a program), cannot be called by either a dynamic call or a bound call. A procedure is the target of a bound call (in RPG the module name and the procedure name may be the same).
- One source member is used to create one RPG module.
- One or more modules can be combined into an ILE program object by using the Create Program (CRTPGM) command.
- Modules created by different high-level languages (HLLs) can be combined into a single program. For example, you could create a program by combining one CL module and one RPG module.
- The same module can be used by several programs.

The Create Bound RPG Program (CRTBNDRPG) command allows you to create a single-module program in one step. This is just the way it was done in OPM. Separate steps occur internally, making a temporary module and then creating a program from the module. The temporary module is deleted as part of the process.

At a high level, what you do is essentially the same as what you did with OPM:

1. Enter source using PDM and specify the source type RPGLE.
2. Use option 14 on the PDM menu to create a program.

A program object is created (assuming there are no severe diagnostics). The fact that the system has internally performed an extra step or two is not noticeable to you.

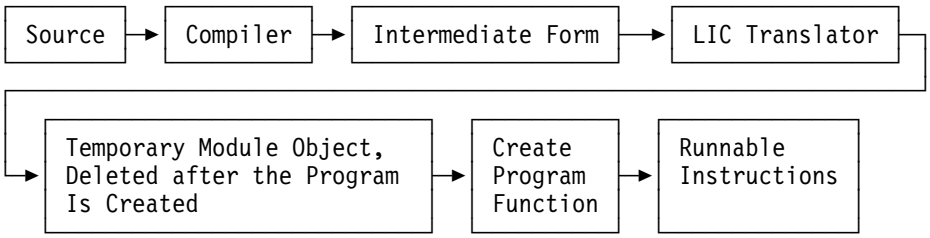

Figure 3-2. Logical View of the Create Bound Program Sequence

How Do You Call the Program?

You use the same technique to call a program that you used before. The call from the CL program operates in the same way. There is no reason to change the CL program because the RPG program is now an ILE program. In our sample, we changed the name of the program; therefore, the CL source had to change. As in the previous example, you can still call an RPG program with a CALL statement in a CL program.

You can try calling the CL program to see how it operates on the new RPG IV program.

```
CALL    PGM(PRTPGMC2) PARM(PRTLIB)
```

The result with the ILE version should look similar to the OPM version. The difference is that two additional programs are listed: PRTPGMC2 and PRTPGMR2. Note that there is no module object in the spooled output. The CRTBNDRPG command creates a temporary module that is deleted when the program is created. You should also note the attributes of the program objects. RPGLE is used, so you can distinguish between OPM and ILE programs.

What Was Accomplished by Converting to RPG IV?

For the purposes of this example, the program source is now in a form in which you can use the new ILE features. This is clearly an advantage. You can also use the new interactive debugger, which can be very helpful.

Considerations Related to Converting to RPG IV

There are certain things to consider when you think about converting your applications to ILE. While converting may be best for some users, for others there may be good reasons not to choose RPG IV at this time:

- You cannot send a program object created with RPG IV source to an earlier release system.
- RPG IV source cannot be sent to someone who does not have the new compiler.
- Once you convert, there is no going back. The conversion program provided only converts RPG III source to RPG IV source.

Should You Convert All Your RPG Source to RPG IV?

It will be somewhat confusing for a programmer to move back and forth between RPG III and RPG IV source. The source looks different, and some functions are specified in a different way.

You may come to the conclusion that it is better to convert all of your RPG source to the new format. You do not have to replace the existing programs. You can convert all of your source in one operation, or you can wait until you need to change the source, and then convert it and replace the programs.

Programmers often need to look at the existing source even though they may not be making any changes to it. Some programmers like the new form of F and D specifications since they are more self-documenting (keywords are used). It may be an advantage, from a readability perspective, to convert to the new format.

There is no specific reason to convert the source unless:

- You want to take advantage of the ILE benefits listed in Chapter 1
- You like the documentation of the new format
- You want consistency (you do not want the confusion of working with two different formats)

Both OPM RPG programs and ILE RPG programs can coexist in the same application. Using RPG IV may be a problem if you ship source or object code to other systems, because RPG IV object programs cannot ship to earlier releases.

If you do not have to consider shipping source or object code to other systems, you can convert to RPG IV in either of these ways:

- All at once
- When you are working on a source member

RPG IV is the RPG standard on the AS/400 system. Therefore, all of the examples you see (and the accompanying education) support the new format. You are making an investment in the future by learning to work with RPG IV.

Chapter 4. Converting CL Source

Roadmap

Steps in example (current step is highlighted):

- A Sample OPM Application
- Converting RPG Source
- **Converting CL Source**
- Making the RPG Program into a Subprogram
- Creating Modules and Binding by Copy
- Making the Subprogram a Service Program

Up to this point, you have been using an OPM CL program to call an ILE RPG program. Now you can convert the OPM CL program to an ILE CL program as well. Be aware that, after you convert to ILE, you can no longer retrieve CL source from the module.

There is no command to convert an OPM CL program to an ILE CL program. Instead, you change the source type specified in the PDM member list from CLP to CLLE. Then, you use option 14 (Create) and PDM generates the correct command. Here are the steps:

1. Start PDM if you are not already there.

   ```
   WRKMBRPDM FILE(PRTLIB/SOURCE)
   ```

2. Change the source type of PRTPGMC2 from CL to CLLE.
3. Change the text description of PRTPGMC2 to `Print program - ILE`.
4. Edit the PRTPGMC2 source. Change the first comment to:

   ```
   /* PRTPGMC2  -  Print program - ILE */
   ```

   The format of the source is the same as it is for OPM CL.
5. End SEU and use PDM option 14 to create the program.

PDM determines the command to use based on the source type. Because the type is CLLE, PDM uses the CRTBNDCL command. The command submitted to batch is:

```
CRTBNDCL   PGM(PRTLIB/PRTPGMC2) SRCFILE(PRTLIB/SOURCE) REPLACE(*YES)
```

No source changes are needed for the sample code. The changes to the CL compiler for ILE primarily affect the internals. The vast majority of the externals (for example, commands and function) remain the same.

In most cases, all you need do is to change the source type to CLLE and re-create the program. For that, you can use PDM option 14. When the program is created, you can try it with:

```
CALL    PGM(PRTPGMC2) PARM(PRTLIB)
```

You should see the same results as with the OPM version. Note that there are no module objects in the spooled output. When you use either of the create bound program commands:

Create Bound RPG Program (CRTBNDRPG)

Create Bound CL Program (CRTBNDCL)

the module object is created only as a temporary object and is deleted when the program is created.

How Compatible Is CL Source?

ILE CL source is very close to 100% compatible with OPM CL source. There are some new ILE features, but if you choose not to use those, most of the source is identical. The known exceptions are the Transfer Control (TFRCTL) and Reclaim Resource (RCLRSC) commands.

TFRCTL This command serves no useful function in an ILE environment. Therefore, it is not supported. You cannot create an ILE CL program with a TFRCTL command in it. To determine which of your programs use TFRCTL, you can use the Print Command Usage (PRTCMDUSG) system command.

RCLRSC This command has a different definition. If you are using it, you should review the new command, Reclaim Activation Group (RCLACTGRP).

There is a new Call Procedure command (CALLPRC) that is used for calling procedures located within modules that are in either the same ILE program or another ILE service program. This is not a conversion problem because the command did not exist for OPM CL; therefore, you cannot use CALLPRC in an OPM CL program. There are several new commands, such as Create Bound CL (CRTBNDCL), that are not restricted. They can appear in either OPM or ILE programs.

Similarly, there are several commands with additional parameters or options, such as the override commands. These are not restricted and can appear in either an OPM or an ILE program, but some of the function makes sense only when operating on ILE objects or functions.

Functions like QCMDEXC and QCMDCHK remain the same. When you use these functions, you use a dynamic call, not a static call. If you write your own commands, there is a potential change to your command definition source.

Does the Create CL Program Function Change?

There are some differences between CRTCLPGM and CRTBNDCL, but most of the function is the same. Let us review what is the same (or logically the same).

The defaults for creating ILE CL programs are similar to those for creating ILE RPG programs. When you specify the create option from the PDM menu, PDM determines which command to use based on the source type. If the type is CLLE, the CRTBNDCL command is submitted to batch, which creates a module as a temporary object and then creates a program by using the module. Here are a couple of significant changes:

- Just as with the RPG compiler, the GENOPT(*LIST) function for creating a listing of the intermediate code is not supported on CRTBNDCL.
- The CL source is no longer stored in the program (or module). This makes the Retrieve CL Source (RTVCLSRC) command unusable against any ILE CL program or module. This means that you must be sure you have an adequate

backup of your CL source, because you can no longer rely on the create command to store the source with the object.

Chapter 5. Making the RPG Program into a Subprogram

Roadmap

Steps in example (current step is highlighted):

- A Sample OPM Application
- Converting RPG Source
- Converting CL Source
- **Making the RPG Program into a Subprogram**
- Creating Modules and Binding by Copy
- Making the Subprogram a Service Program

Related discussion:

- Integrated Language Environment

This chapter explains how to create simple subprograms in ILE by showing how to make an RPG subprogram from the date conversion subroutine. The purpose of this step is to allow us to show how to use the subprogram function as:

- A separate module in a multiple-module program
- A separate module in a service program

Making a Subprogram of the Date Conversion

Figure 5-1 shows the result of converting the date subroutine into a separate subprogram.

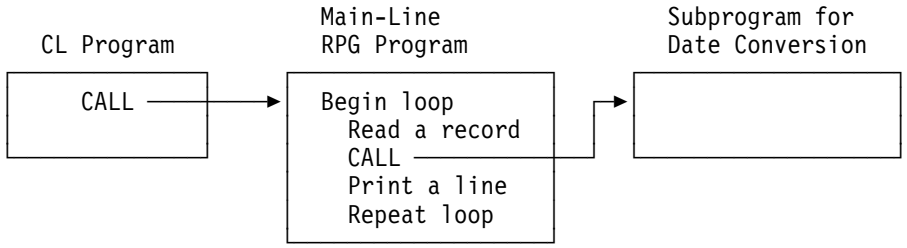

Figure 5-1. Date Subroutine Converted to Program

Some application designs look very similar to this one. A subprogram is created for a frequently used function that can be called from many different main-line programs.

Creating the CL Program

The first series of steps creates a different CL program.

1. Use the PDM copy option to copy the source from the PRTPGMC2 member to a new member PRTPGMC3.
2. Change the text description of PRTPGMC3 to `Print program - ILE with subprogram`.
3. Use PDM option 2 to edit the PRTPGMC3 source member. Change the comment on line 00.10 to:

   ```
   /* PRTPGMC3 - Print program - ILE with subprogram   */
   ```

4. Change the CALL statement to:

```
CALL       PGM(PRTPGMR3) PARM(&LIB)
```

5. End SEU. Use PDM option 14 to create an ILE CL program.

Creating the ILE RPG Subprogram

1. Use PDM F6 to create a new member. Name the new member DATCVT, assign it a source type of RPGLE, and specify a text description of `Date conversion ILE program`.
2. Enter the DATCVT source as shown in Figure 5-2.

   The numbers (for example, 00.10) at the left margin of the example are line numbers. They are entered for you automatically.

```
00.10      * DATCVT - Date conversion ILE program
00.20      *
00.30      * Convert date from MMDDYY to YYMMDD format
00.40      *
00.50     C     *ENTRY        PLIST                                          Parm list
00.60     C                   PARM                    MMDDYY         6       MMDDYY fmt
00.70     C                   PARM                    YYMMDD         6       YYMMDD fmt
00.80     C                   PARM                    SETLR          4       Set LR
00.90      *
01.00     C                   MOVE      MMDDYY        WORK2          2       Move YY
01.10     C                   MOVEL     WORK2         YYMMDD         6       Move YY
01.20     C                   MOVEL     MMDDYY        WORK4          4       Move MMDD
01.30     C                   MOVE      WORK4         YYMMDD                 Move MMDD
01.40      *
01.50     C     SETLR         IFEQ      '*YES'                               If *YES
01.60     C                   SETON                                      LR  Set LR
01.70     C                   ENDIF                                          If *NO
01.80     C                   RETURN                                         Return
```

Figure 5-2. Source for ILE RPG Date-Conversion Program

The following is a brief explanation of the code:

- Lines 00.50-00.80

  The program accepts a parameter list of the two date formats. The MMDDYY field is input, and the YYMMDD field is output. The SETLR parameter tells the program what to do when it returns.

  Even with ILE programs, if you are going to call the same program repeatedly, it is a better performance approach to return with LR off. The same concept is true whether it is a stand-alone program (as this example is) or a module.

- Lines 01.00-01.30

  The program converts the date format. This is the same code that was in the subroutine in the prior examples.

- Lines 01.50-01.80

  The program returns with LR on or off. By returning with LR off, RPG does not have to go through its initialization routine the next time the program is called. In a program of this size, the initialization routine is not very long, but in a more realistic subprogram, you could be opening files, or you could have many more fields that have to be initialized.

  A traditional parameter list is being passed between the two programs.

3. End SEU and use PDM option 14 to create the program DATCVT.

Creating the Main-Line ILE RPG Program

The next series of steps creates the main-line ILE RPG program, PRTPGMR3. If you are coding these examples on a system as you read through this book, follow these steps:

1. Enter the source for program PRTPGMR3 as shown in Figure 5-3.

```
00.10      * PRTPGMR3 - Print program - ILE with subprogram
00.20      *
00.30      * QADSPOBJ is the output from DSPOBJD - Override occurs in CL
00.40     FQADSPOBJ  IF   E             DISK
00.50     FQPRINT    O    F  132        PRINTER OFLIND(*INOF)
00.60      ****************************************************************
00.70      * Parameter lists
00.80     C     *ENTRY        PLIST                                          Parm list
00.90     C                   PARM                    LIB             10     Library
01.00     C     PLIST1        PLIST
01.10     C                   PARM                    MMDDYY           6     MMDDYY fmt
01.20     C                   PARM                    YYMMDD           6     YYMMDD fmt
01.30     C                   PARM                    SETLR            4     Set LR sts
01.40      *
01.50     C                   EXCEPT    HDG                                  Prt heading
01.60      ****************************************************************
01.70      * Read a record
01.80      * QLIDOBJD is the format name of the QADSPOBJ file
01.90     C                   READ      QLIDOBJD                       20    Read
02.00      * Continue reading until EOF
02.10     C     *IN20         DOWEQ     '0'                                  Not EOF
02.20      ****************************************************************
02.30      * Use a program to convert the date from MMDDYY to YYMMDD
02.40     C                   MOVE      ODUDAT        MMDDYY           6     MMDDYY fmt
02.50     C                   CALL      'DATCVT'      PLIST1                 Convert date
02.60     C                   MOVE      YYMMDD        LSTUSD           6 0   Last used dt
02.70     C                   EXCEPT    DETAIL                               Print detail
02.80     C   OF              EXCEPT    HDG                                  Prt heading
02.90     C                   READ      QLIDOBJD                       20    Read
03.00     C                   ENDDO                                          Loop Back
03.10      * End the program
03.20     C                   MOVE      '*YES'        SETLR                  Set LR
03.30     C                   CALL      'DATCVT'      PLIST1                 Last CALL
03.40     C                   SETON                                      LR  Set LR
03.50      ****************************************************************
03.60     OQPRINT    E            HDG            2 06
03.70     O                                          25 'Objects '
03.80     O                                             'in Library - '
03.90     O                       LIB
04.00     O          E            HDG            2
04.10     O                                           6 'Object'
04.20     O                                          18 'Obj type'
04.30     O                                          30 'Attribute'
04.40     O                                          42 'Last used'
04.50     O          E            DETAIL         1
04.60     O                       ODOBNM             10
04.70     O                       ODOBTP             19
04.80     O                       ODOBAT             33
04.90     O                       LSTUSD        Y    41
```

Figure 5-3. Source for Main-Line ILE RPG Program PRTPGRM3

The program continues to use a subroutine to convert the date. The end-of-program routine also calls the subroutine but first moves the value *YES into the SETLR field. This value will be used to end the subprogram.

Another alternative might be to free the space used by any such programs with a reclaim command run in the CL program. The reclaim commands are Reclaim Resource (RCLRSC) and Reclaim Activation Group (RCLACTGRP).

In this application, the subprogram is called when the main program decides to end. Setting LR on in the subprogram properly closes any files that are open (none in this example) and frees up the programs' work areas.

2. Create the main-line RPG program by using PDM option 14.
3. Call the CL program:

```
CALL    PGM(PRTPGMC3) PARM(PRTLIB)
```

At this point, we have not bound any of the functions together. They use the standard dynamic call function. You should see the same result that you did before, except that there are now seven programs in the library. Note again that there are no module objects at this point. Modules were created by each of the ILE create commands, but the objects were temporary and were deleted when the program objects were created.

Is There a Performance Advantage at This Point?

If only one person uses the subprogram, there is no performance advantage. The subprogram that you are calling is very small, and there is overhead to call the subprogram for each record read. This approach makes sense if you can move a significant piece of logic to a subprogram, thus minimizing overhead.

If many applications are active and calling the same subprogram simultaneously, more efficiency might be needed. Additional efficiency is available if the system has one copy of the code in main storage, rather than having the same routine in every program. Having the same routine in every program uses more storage. The total amount of main storage used by all applications influences performance. Auxiliary storage use is also a consideration.

For more typical use, this is a very complex question with significant variance in the answer, depending on the specific environment. As main storage size increases, there is less contention for main storage. As a result, the probability diminishes that a subprogram approach is more efficient. Performance is not the only question in application design. Maintenance is also a good reason for wanting only one copy of an object. If a problem needs to be corrected, it is best to update only one copy of the object rather than many. Another consideration is how much disk storage you use.

Chapter 6. Creating Modules and Binding by Copy

Roadmap

Steps in example (current step is highlighted):

- A Sample OPM Application
- Converting RPG Source
- Converting CL Source
- Making the RPG Program into a Subprogram
- **Creating Modules and Binding by Copy**
- Making the Subprogram a Service Program

Related discussions:

- Integrated Language Environment
- Modularity
- Updating

This chapter focuses on the binding of modules in the same and different languages. It also explains how to create modules and bind by copy.

Binding the Two RPG Functions into a Single Program

Figure 6-1 shows the two modules in the RPG program bound together.

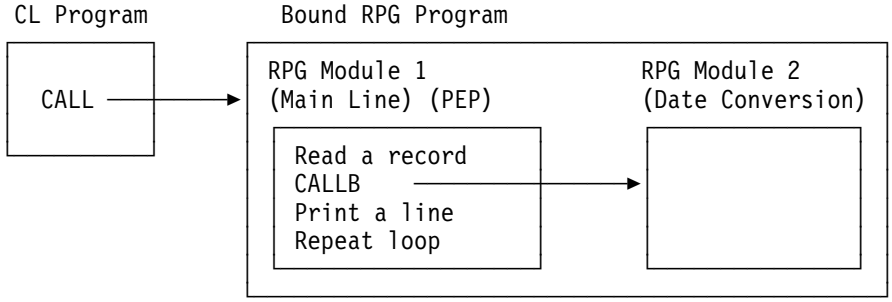

Figure 6-1. Two RPG Modules Bound Together

The CL program continues to use CALL, rather than CALLPRC. Whenever you call a program object (not a procedure) from CL, you always use CALL and not CALLPRC.

The RPG code must use the new operation code CALLB. This is a static call, which is used when other procedures are called.

In these steps, a module is created as a separate object type (*MODULE). An RPG module is created by using the CRTRPGMOD command. A module object cannot be run directly. You must use the CRTPGM command to bind module objects into a program object.

Creating the ILE RPG Main-Line Module

If you are coding these examples on a system as you read through this book, follow these steps:

1. Copy the source from PRTPGMR3 to a new member named PRTPGMR4.
2. Change the text description to `Print program - ILE main module`.
3. Use PDM option 2 to edit the PRTPGMR4 source member.
4. Change the text comment to:

   ```
   F* PRTPGMR4 - Print program - ILE main line module
   ```

5. Change the CALL operation on lines 02.50 and 03.30 to CALLB, and specify the name of the module (procedure DATCVT2). DATCVT2 is the date conversion module that you create next.

   ```
   C           CALLB     'DATCVT2'    PLIST1        Date convert
   ```

6. Use PDM option 15 to create a module. Because option 15 is chosen instead of option 14, PDM submits the CRTRPGMOD command rather than the CRTBNDRPG command used before. The command submitted is:

   ```
   CRTRPGMOD  MODULE(PRTLIB/PRTPGMR4) SRCFILE(PRTLIB/SOURCE)
   ```

Creating the ILE RPG Subprogram Module

1. Copy the source from DATCVT to a new member DATCVT2.
2. Change the text description to `Date convert - MDY format to YMD`.
3. Use PDM option 2 to edit the DATCVT2 source member.
4. Change the text comment to:

   ```
   * DATCVT2 - Date conversion - MDY format to YMD
   ```

5. Use PDM option 15 to create a module. The command submitted is:

   ```
   CRTRPGMOD  MODULE(PRTLIB/DATCVT2) SRCFILE(PRTLIB/SOURCE)
   ```

6. At this point the two RPG modules exist and can be bound into a single program. The CRTPGM command is used to create a program from one or more modules. No matter which ILE language the modules were written in, CRTPGM is used.

PDM Option for CRTPGM Command

Since CRTPGM uses modules and service programs (not source members) as input, there is no option on the Work with Members Using PDM display to submit a CRTPGM command. The CRTPGM command must be entered on the command line. The Work with Objects Using PDM display does include option 26 (Create program), but to use that option you must have created a *MODULE object.

Even if the object already exists, when you use PDM, you cannot take the defaults as you normally do for creating a simple OPM or ILE program. Although there are defaults for all but the PGM parameter, even a two-module program (as in our sample) does not create properly if you use the defaults. For many of you, the CRTPGM command requires a change in the mechanics of your usual way of operating.

There are a few important parameters on the CRTPGM command:

PGM This is the name you want to assign to the object. (See discussion later in this chapter.)

MODULE This is a list of the module names. There is no option to use the existing ones. You have to name the list each time. The default is *PGM, which provides a solution only if you have a single-module program.

To take advantage of the ENTMOD parameter, you list the first module as the program entry procedure (PEP). Other alternatives to a module list are to use generic names or a binding directory. For information about binding directories, see “Binding Directory” on page 14-11.

ENTMOD The entry module contains the program entry procedure (PEP). This is the module whose PEP gets control when the program is called. This is the only module whose PEP is used by the binder. The PEPs of all of the other modules are not reachable; the code in those PEPs is not run at all. The value for the ENTMOD parameter defaults to *FIRST, which means that the first module that contains a PEP is used as the PEP for the program object. Another value you can specify for the ENTMOD parameter is *PGM. This means that a module by the same name as the program being created is used as the entry point when the program is called. This module contains a PEP. The *PGM value may be quite beneficial for RPG users.

ACTGRP This parameter defaults to *NEW, which causes a new activation group to be created. You should consider using a named activation group, such as QILE, which is the default on CRTBNDRPG if DFTACTGRP(*NO) is specified, rather than specifying *CALLER. When *CALLER is specified, the program runs in the activation group from which it was called. This can be a problem if the program is called by an OPM program and runs in the default activation group.

Assume that you decide to create a new program object that contains modules PRTPGMR4 and DATCVT2. Also assume that you want to name the new program object PRTPGMI. To do so, you would enter the command:

```
CRTPGM   PGM(PRTPGMI)
         MODULE(PRTPGMR4 DATCVT2)
         ACTGRP(*CALLER)
```

The CRTPGM command:

- Names the new program
- Names the two modules
- Specifies which module is the entry point (in this case, the main-line RPG module, PRTPGMR4)
- Specifies in which activation group the program runs (in this case, the activation group from which it is called)

If you need to create this program again, consider creating a CL program that uses CRTPGM for maintenance. That way, you do not have to specify the CRTPGM command parameter values manually each time.

Creating the CL Program

If you are coding these examples on a system as you read through this book, follow these steps.

1. Use the PDM copy option to copy the source from the PRTPGMC3 member to a new member named PRTPGMC4.
2. Change the text description to `Print program - ILE (calls pgm with bound modules)`.
3. Use PDM option 2 to edit the PRTPGMC4 source member. Change the first comment to:

   ```
   /* PRTPGMC4 - Print program - ILE calls pgm with bound modules */
   ```

4. Change the CALL statement to the name of the bound program:

   ```
   CALL       PGM(PRTPGMI) PARM(&LIB)
   ```

5. Using PDM option 14, create the program.
6. Call the new program:

   ```
   CALL      PGM(PRTPGMC4) PARM(PRTLIB)
   ```

You should see a result similar to the OPM result, except that the library contains more programs and two additional entries with an object type of *MODULE. When you use the CRTRPGMOD command, you see the objects with an object type of *MODULE. When you use the CRTBNDxxx command, you do not see *MODULE objects.

The module object is not of any value at this point unless you need to re-create the program or create another one using it. When you call the program, the module object is not used. The entity that is used is the copy of the module that has become part of the PRTPGMI program. Within the PRTPGMI program is the same set of instructions that exist within the modules. You can consider it to be a duplicate within the program.

Binding CL and RPG Modules into a Single Program

The next step is to make a program object consisting of the two RPG modules and the one CL module (see Figure 6-2).

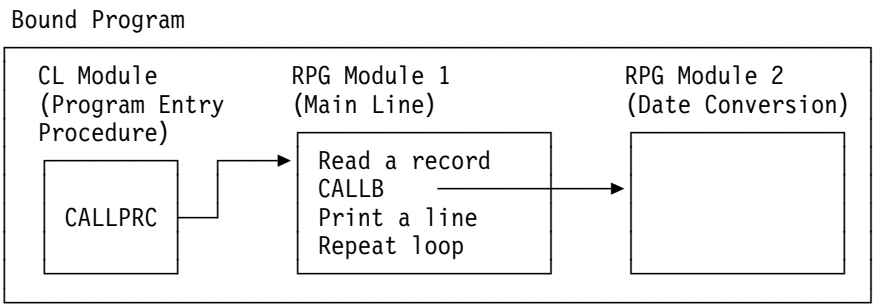

Figure 6-2. A Sample Bound Program

Changing the CL Program

The CL code has to change to use a Call Procedure (CALLPRC) command. This is the bound call in CL that calls a procedure within a module. You cannot use the CALL command because the target of the CALL command is a program object, not a procedure. The command has to be CALLPRC.

The next series of steps completes the process of preparing the CL program to be bound. If you are coding these examples on a system as you read through this book, follow these steps:

1. Copy the source from the member PRTPGMC4 to a new member named PRTPGMC5.
2. Change the text description to `Print program - ILE CL entry`.
3. Use PDM option 2 to edit the program PRTPGMC5. Change the text comment to:

   ```
   /* PRTPGMC5 - Print program - ILE CL entry    */
   ```

4. Change the call command to:

   ```
   CALLPRC PRC(PRTPGMR4)
   ```

5. Use PDM option 15 to create the PRTPGMC5 module object. PDM recognizes the type CLLE and generates the CRTCLMOD command to be submitted to batch. The command looks like this:

   ```
   CRTCLMOD   MOD(PRTLIB/PRTPGMC5)   SRCFILE(PRTLIB/SOURCE)
   ```

   When the batch job completes, the CL module is ready to be bound into a program.

Creating the Bound Program

The next series of steps creates the bound program.

1. Use the CRTPGM command to create the bound program (with a name of PRTPGMI2) consisting of the following modules: PRTPGMC5, PRTPGMR4, and DATCVT2. Once again, PDM does not know what values you want to specify for the required parameters, so you are prompted for the Create Program (CRTPGM) command. You need to list PRTPGMC5 first on the MODULE parameter. This allows the ENTMOD parameter to default to the CL module that contains the PEP. You should consider using a named activation group, rather than specifying *CALLER. When *CALLER is specified, the program runs in the activation group from which it was called. This can be a problem if the program is called by an OPM program and runs in the default activation group. You should enter the command as follows:

   ```
   CRTPGM   PGM(PRTLIB/PRTPGMI2)
            MODULE(PRTLIB/PRTPGMC5
                   PRTLIB/PRTPGMR4
                   PRTLIB/DATCVT2)
            ACTGRP(*CALLER)
   ```

2. Call program PRTPGM12:

   ```
   CALL   PGM(PRTPGMI2) PARM(PRTLIB)
   ```

   You should see a result similar to the OPM result, except that there are more programs in the library and the CL module exists. Note that the attribute of the PRTPGMC5 module is CLLE. The attribute of the PRTPGMI2 program is

CLLE because the attribute field assumes that the attribute of the module specified has the program entry procedure (in this case, PRTPGMC5).

3. Use the Display Program (DSPPGM) command to look at the program description:

   ```
   DSPPGM  PGM(PRTPGMI2)
   ```

   The results should look similar to Figure 6-3.

```
                                Display Program Information

Program  . . . . . :   PRTPGMI2                          Library  . . . . . :  PRTLIB
Owner  . . . . . . :   XYZ
Program attribute  . . :   CLLE
Detail . . . . . . . . :   *BASIC

Program creation information:
  Program creation date/time . . . . . :   06/27/XX  09:48:12
  Type of program  . . . . . :   ILE
  Program entry procedure module . . . . . :   PRTPGMC5
    Library  . . . . . :     PRTLIB
  Activation group attribute . . . . . :   *CALLER
  Shared activation group  . . . . . :   *NO
  User profile . . . . . :   *USER
  User adopted authority . . . . . :   *YES
  Coded character set identifier . . . . . :   65535
  Number of modules  . . . . . :   3
  Number of service programs . . . . . :   4
  Number of unresolved references  . . . . . :   0
  Number of copyrights . . . . . :   0
  Observable information compressed  . . . . . :   *NO
  Run time information compressed  . . . . . :   *NO
  Allow update . . . . . :   *YES
  Text description . . . . . :   PRTPGMC5 - Print program - ILE CL entry
Program statistics:
  Number of parameters . . . . . :   0            255
  Associated space size (decompressed) . . . . . :   1408
  Static storage size  . . . . . :   7280
  Allow static storage reinitialization  . . . . :   *NO
  Program size . . . . . :   181248
  Static state   . . . . . :   *USER
  Program domain . . . . . :   *USER
  Release program created on . . . . . :   V3R1M0
  Release program created for  . . . . . :   V3R1M0
  Earliest release program can run . . . . . :   V3R1M0
Program performance information:
  Paging pool  . . . . . :   *USER
  Paging amount  . . . . . :   *BLOCK
```

Figure 6-3. Display Program Information for Program PRTPGMI2

Additional information about an ILE program (such as a list of modules) is available by using the Display Program (DSPPGM) command.

Performance Discussion

This example shows a negligible improvement in performance compared to the previous example, in which two RPG programs are bound together in a program. You substituted one bound call for one dynamic call each time you call the application.

However, at this point, the user of the program cannot call the RPG functions directly. The only program that can be called is PRTPGMI2. The application has only a single part. If you are going to ship the application as object code, this can be a significant advantage.

Chapter 7. Making the Subprogram a Service Program

Roadmap

Steps in example (current step is highlighted):

- A Sample OPM Application
- Converting RPG Source
- Converting CL Source
- Making the RPG Program into a Subprogram
- Creating Modules and Binding by Copy
- **Making the Subprogram a Service Program**

Related discussions:

- Integrated Language Environment
- Service Programs

You have already created a bound program with three modules, one CL and two RPG. Figure 7-1 shows the bound program with which we start.

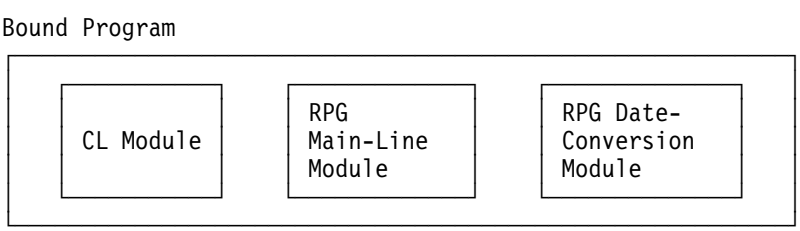

Figure 7-1. Bound Program Containing a Date-Conversion Module

Now we will make the date-conversion module of this program part of a service program. A service program is a collection of runnable procedures and available data items that can be accessed easily and directly by other ILE programs or service programs. In many respects, a service program is similar to a subroutine library or procedure library.

Service programs provide common services that other ILE objects may need. An example of a set of service programs provided by OS/400 is the run-time procedures for a language. These run-time procedures often include such items as mathematical procedures and common input/output procedures.

In an application with a minimal number of date conversions, you are unlikely to use a service program approach if you are interested in optimizing. However, we will make the date conversion module part of a service program here to show an example of how to create service programs.

Normally, a service program has more than a single module (procedure). Therefore, a second module is added to the service program in this example. The first module converts a date from the month-day-year (MDY) format to the year-month-day (YMD) format. The second module converts a date from the YMD format to the MDY format. The user of the service program must select which procedure to use and the variables that are passed back and forth.

In this simple example, the procedures within the service program do not communicate with each other. Instead, the calling programs identify which service function they want and they call these procedures directly.

In the same manner that a multiple-module ILE program communicates, you can arrange for communication between a program and a service program by:

- Using a parameter list
- Passing import and export variables (for information about imports and exports, see “Imports and Exports” on page 9-6)
- Using objects, such as the local data area, database files, message queues, and so on

Use of a Service Program in a Sample Application

If you are coding these examples on a system as you read through this book, here are the steps for completing the sample application.

Creating the Date-Conversion Modules

The simple service program that is used as an example has two modules:

MDYYMD	Converts from month-day-year format to year-month-day format
YMDMDY	Converts from year-month-day format to month-day-year format

The first set of steps creates the modules.

1. Use PDM function key F6=Create to add a new member named MDYYMD. Use type RPGLE, and use the text description `Date conversion module - MMDDYY format to YYMMDD`.
2. When the SEU Edit display appears, use function key F15=Browse/Copy to copy the source from the DATCVT2 module. The DATCVT2 source already uses a parameter list approach, so you need to change only the comment.
3. Change the first comment to

   ```
   * MDYYMD - Date conversion - parms - MMDDYY format to YYMMDD
   ```

4. End SEU and use PDM option 15 to create the module.
5. Use PDM to add a new member named YMDMDY. Make it type RPGLE, and use the text description `Date conversion module - parms - YYMMDD format to MMDDYY`.
6. Enter the following code:

```
00.10      * YMDMDY - Date conversion module - YYMMDD format to MMDDYY
00.20      *
00.30      * Convert date from YYMMDD to MMDDYY format
00.40      *
00.50     C     *ENTRY        PLIST                                          Parm list
00.60     C                   PARM                    YYMMDD         6       YYMMDD fmt
00.70     C                   PARM                    MMDDYY         6       MMDDYY fmt
00.80      *
00.90     C                   MOVEL     YYMMDD        WORK2          2       Move YY
01.00     C                   MOVE      WORK2         MMDDYY         6       Move YY
01.10     C                   MOVE      YYMMDD        WORK4          4       Move MMDD
01.20     C                   MOVE      WORK4         MMDDYY                 Move MMDD
01.30      *
01.40     C                   RETURN                                         Return
```

7. End SEU and use PDM option 15 to create the module.

Creating the Date-Conversion Service Program

The two modules now exist and can be combined into a service program with the Create Service Program command:

```
CRTSRVPGM   SRVPGM(DATSRV1) MODULE(MDYYMD YMDMDY) EXPORT(*ALL)
            TEXT('Date Conversion Service Program')
```

The EXPORT(*ALL) parameter means that all variable and procedure names exported from the modules used to create the service program are also exported from the service program. You can use the Display Module (DSPMOD) command to see what is exported from a module object. In this case, there are no identified variables to be exported because a parameter list is used. This means that the only exports known to the DATSRV1 program are the procedure names.

PDM Option for CRTSRVPGM

There is no option on the Work with Member Using PDM display for creating a service program. Like CRTPGM (see “PDM Option for CRTPGM Command” on page 6-2), CRTSRVPGM uses modules and service programs (not source members) as input, so there is no option on the WRKMBRPDM display to create a service program. A PDM option does exist from the WRKOBJPDM display, but as with CRTPGM, you cannot use the option and take the defaults as you can with PDM options for OPM and CRTBNDxxx. (See Appendix A, “Packaging CRTPGM and CRTSRVPGM” on page A-1 for recommendations.)

Creating the Calling ILE CL Module

Assume that you have decided to combine the CL function and the RPG main-line function into a single ILE program. If you are coding these examples on a system as you read through this book, follow these steps.

1. Use PDM function key F6=Create to add a new member named PRTPGMC7. Make it type CLLE, and use the text description `Print program - uses service pgm with parms`.
2. When the SEU Edit display appears, use function key F15=Browse/Copy to copy the source from the PRTPGMC5 source.
3. Change the first comment to:

   ```
   /* PRTPGMC7 - Print program - service pgm with parms  */
   ```

4. Change the CALL command to:

   ```
   CALLPRC   PRC(PRTPGMR7) PARM(&LIB)
   ```

5. End SEU and use PDM option 15 to create the module.

Creating the ILE RPG Module

1. Use PDM function key F6=Create to add a new member named PRTPGMR7. Make it type RPGLE, and use the text description `Print program - uses service pgm with parms`.
2. When the SEU Edit display appears, use function key F15=Browse/Copy to copy the source from the PRTPGMR4 source.
3. Change the first comment to:

   ```
   * PRTPGMR7 - Print program - service pgm with parms
   ```

4. Change the CALL operation to:

```
CALLB   'MDYYMD'   PLIST1
```

5. End SEU and use PDM option 15 to create the module.

Creating the ILE Program

1. Create the program object PRTPGMI3 by using the following command:

   ```
   CRTPGM   PGM(PRTPGMI3) MODULE(PRTPGMC7 PRTPGMR7)
            ENTMOD(PRTPGMC7) BNDSRVPGM(DATSRV1)
            ACTGRP(*CALLER)
            TEXT('Print program - uses serv pgm with parms')
   ```

2. Call the program

   ```
   CALL    PGM(PRTPGMI3) PARM(PRTLIB)
   ```

3. Display the spooled file that was produced. You see more programs and modules than before. For the first time, you see a *SRVPGM object type. Note the attribute of the service program. All of the modules in the *SRVPGM have an attribute of RPGLE, so the resulting *SRVPGM also has an attribute of RPGLE. If the service program contained modules written in more than one language, the attribute would be blank.

   You cannot use DSPPGM to look at a service program. Instead, you use the command DSPSRVPGM:

   ```
   DSPSRVPGM   SRVPGM(DATSRV1)
   ```

In the upper right corner of the Display Service Program screen, you see `Display 1 of 10`. Display 1 lists the basic information about the service program. Each time you press the Enter key, the display number increases by 1. You see additional information, such as the size of the service program, the list of modules that were bound by copy into the service program, the list of service programs that were bound by reference (usually the run-time routines), procedures that are exported, and the associated signature.

Discussions

Chapter 8. Original Program Model

This chapter focuses on aspects of OPM that are affected by ILE.

Original Program Model Discussion

To better understand the effect of ILE on your applications, it is useful to look at the original program model and make certain comparisons. Certain aspects of OPM are changed more than others. This discussion focuses on:

- Program size
- Program objects as read-only code
- Program stack
- Activated programs
- Exception handling

Program Translation and Size in OPM

In this example there are two programs, PRTPGMC (CL) and PRTPGMR (RPG).

Table 8-1. Sizes of Program Objects. This example shows the size of the program objects in this release.

Program	Size
PRTPGMC	9 216 bytes
PRTPGMR	33 280 bytes

When a program is created in the original program model, the HLL compiler generates an intermediate form of code, which is then passed to a common system component called the program resolution monitor. The program resolution monitor converts the intermediate form to a different form called the program template.

This program template is then passed to the Licensed Internal Code function known as the translator. The translator translates the program template into machine-runnable instructions (see Figure 8-1).

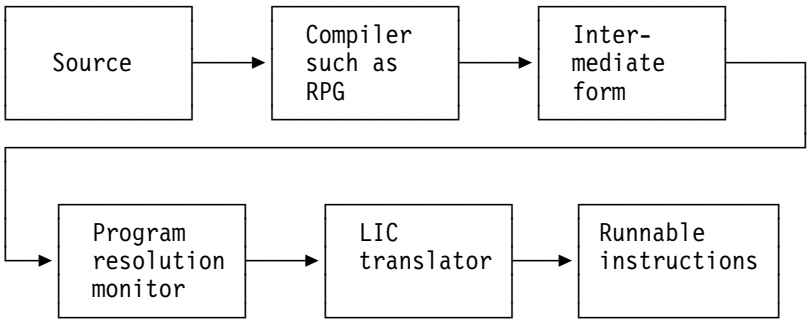

Figure 8-1. Process for Compiling and Translating an OPM Program

OPM Program Objects

The program object contains not only the runnable instructions, but also the program template. The presence of the program template is also called observability.

Observability allows your program to be debugged. You can remove the observability section from the program to save space. However, if you do so, you lose the debugging information (including the formatted dump) and the capability to allow the system to retranslate (re-encapsulate) by using just the program template.

You can remove the observability of an OPM program by using the RMVOBS parameter on the Change Program (CHGPGM) command. For example:

```
CHGPGM   PGM(PRTPGMC) RMVOBS(*YES)
```

An example of the use of the program template occurs in the migration from System/38* to AS/400. When System/38 program objects that contain the program template are restored on the AS/400, they are retranslated. If you delete the program template of a S/38 program, it cannot be restored on the AS/400. You have to re-create the program from source to run it on the AS/400.

OPM Program Template

If you remove the observability, you may have to re-create a program from source in order to move to new hardware. The program template is also used for internal retranslation functions, such as changing the USRPRF option of an existing program by using the CHGPGM command. If you remove the observability of the program, then CHGPGM cannot be used. You have to re-create the program from source.

In OPM programs, the program template and the debug information are logically separate but interwoven pieces. You cannot get rid of only one of the pieces. In ILE programs, this concept changes and you can remove either one or both pieces.

By default, CL programs also store the CL source (the comments are dropped) in the program object (see Figure 8-2 on page 8-3). You can access the source from the program object by using the Retrieve CL Source (RTVCLSRC) command. If you do not want the source stored in the program object, you have to specify ALWRTVSRC(*NO) when you create the program:

```
CRTCLPGM  ...  ALWRTVSRC(*NO)
```

The RTVCLSRC command can be very useful if you lose your source. However, many programmers do not rely on this capability because the source is reformatted (it does not look the way it was keyed) and all of your comments are lost. In the current ILE release, the CL source is **not** stored with the program, and the RTVCLSRC command cannot be used.

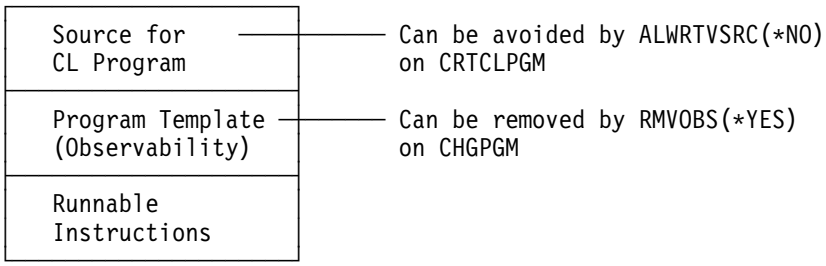

Figure 8-2. An OPM Program Object

Sizes of OPM Programs

Table 8-2 is an example of the programs that were created and what these options mean to the size of the programs in Version 3 Release 1.

Table 8-2. Program Sizes with Various Program Options. These examples show the effect of various options on the size of the programs in Version 3 Release 1.

Program	Original Size	Eliminating CL Source	Removing Observability	Removing Observability and Eliminating CL Source
PRTPGMC	9216	9216	4096	4096
PRTPGMR	33280		9728	9728

The observability portion of a program represents a significant amount of storage. You can reduce the size by using the Compress Object (CPROBJ) command and still retain the benefit of observability when you need it. The CPROBJ command compresses the observability. Thus, the system automatically decompresses the observability if it needs the data.

You can request that the entire program object be compressed, or just the observability portion. If you use the debug capability, the observability is then decompressed and the object returns to its original size in the system. You can use the CPROBJ command periodically to squeeze the size back down again.

Table 8-3 is an example of how the CPROBJ PGMOPT(*OBS) command affects programs PRTPGMC and PRTPGMR.

*Table 8-3. Program Sizes Using CPROBJ PGMOPT(*OBS). This example shows the effect of using the CPROBJ PGMOPT(*OBS) command on the size of the programs in Version 3 Release 1.*

Program	Original Size	Compressing Observability	Removing Observability
PRTPGMC	9216	6656	4096
PRTPGMR	33280	22016	7728

The Change Program (CHGPGM) command also provides an OPTIMIZE option (assuming the program template still exists) that can provide a smaller object size and faster processing. Generally, the OPTIMIZE option causes these effects only

in large programs. It has very little effect on CL programs, which consist mostly of calls to CL commands (each calling its own command-processing program).

Table 8-4 is an example of how the CHGPGM OPTIMIZE(*YES) command affects programs PRTPGMC and PRTPGMR. Later, we compare these data to ILE versions of these same programs.

*Table 8-4. Program Sizes Using CHGPGM OPTIMIZE(*YES). This example shows the effect of using the CHGPGM OPTIMIZE(*YES) command on the size of the programs in Version 3 Release 1.*

Program	Original Size	OPTIMIZE(*YES)
PRTPGMC	9216	8704
PRTPGMR	33280	33280

OPM Program Objects Are Read-Only Code

When a program object is created on the AS/400, the code is read only. This means that multiple users can use the same program, although only one copy of the run instructions exists in main storage. Each user of the same program has a unique work space in which the variable information is kept (see Figure 8-3).

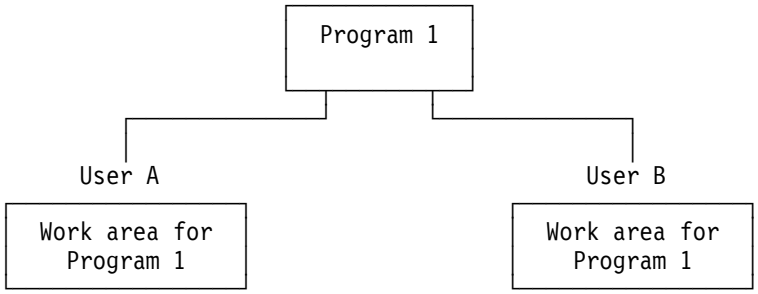

Figure 8-3. Both User A and User B Calling the Same Program

The work area for a single user is placed in the process access group (PAG). This fact does not change with ILE.

Call Stack in OPM

The call stack is the concept used by the system within a given job to determine which program is active. In this simple application, the CL program PRTPGMC calls the RPG program PRTPGMR (see Figure 8-4).

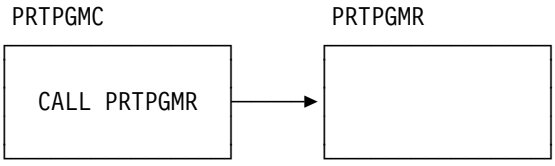

Figure 8-4. CL Program Calling RPG Program

When a program is called, you can think of the program as being logically added to the bottom of the call stack. At that time, control is transferred to it. When a program ends, it is taken out of the call stack and control returns to the previous program. There may be many programs in the call stack, but only the last program is active.

If you look at the call stack by using the Display Job (DSPJOB) command, you normally see IBM programs ahead of PRTPGMC. During the running of PRTPGMR, you would normally see other IBM programs that were called by PRTPGMR and that perform certain data management or RPG functions. The concept of the call stack also exists in ILE, but there are changes that are discussed in “Call Stack” on page 9-5.

Open Files and RCLRSC in OPM

An RPG program can return with the LR indicator off. This means that the work areas associated with the program (for example, variables and open files) remain as they were in the process access group, waiting for the program to be called again.

CL Programs do not have this capability. Every call to a CL program causes the work areas to be reinitialized.

When an RPG program returns with LR off, it is no longer in the call stack. However, the system knows that the program is already activated.

If you never call the program again, the work areas remain intact and the files remain open until your job ends. You can close the files and clean up the work areas by using the Reclaim Resource (RCLRSC) command.

OPM Exception Handling

The RPG example program makes only one exception test, which is for an end-of-file condition. The RPG program has significant built-in exception handling. If you call the RPG program directly, you see an example of this with the RPG1216 inquiry message.

Most of the exception handling in ILE RPG is the same as it was in OPM RPG. There are some new functions that you may be interested in, which are described in the *ILE RPG/400 Programmer's Guide*.

The CL example program monitors for one error condition. It relies on the standard exception-handling routines that are part of any CL program. The concept of monitoring for messages by using the MONMSG command is continued with ILE CL programs. For the most part, you will find additional function to assist you in writing more resilient applications (those with code that makes them less likely to fail).

Original and Extended Program Models

The original program model (OPM) is used by the following languages:

CL
RPG
COBOL
PL/I
BASIC

Another program model, called the extended program model (EPM), supports the following languages:

C
Pascal

FORTRAN

Unlike OPM, which allows calls only to a program as a whole, the extended program model allows calls to specific procedures within a program. (A procedure is a set of self-contained, high-level language statements that perform a particular task and return control to the caller.)

Except for allowing multiple entry points to a program, EPM functions are very similar to OPM functions. Therefore, unless specifically noted, this book uses the term original program model (OPM) to mean both OPM and EPM.

Calling RPG and CL Programs in OPM

In OPM there is no direct method of preventing a user from calling a subprogram, either as a mistake or on purpose. Later, we will see how ILE prevents such a problem.

In the spooled output, any objects with a last-used date of 0/00/00 have never been used. In our example, the system marks the CL program as used as soon as you call it. During the running of PRTPGMC, an output file consisting of the various objects in PRTLIB (the library name passed as the parameter) is created. At the time the output file is created, the CL program has been used, and the listing reflects this usage.

When PRTPGMC is first run, the RPG program is shown as never having been used. This is because the program is not yet called at the time the DSPOBJD command is run. If you run the program again, the RPG program is displayed as used.

How Many Calls Are Run during This Application?

The number of calls run during this application is not just one (PRTPGMR) or two (PRTPGMC and PRTPGMR). In addition to the CALL commands that you coded, the system performed many calls to run this application.

For example, when the call is made to the CL program, the system performs many internal calls for the CL commands that you specified. It accesses the command definition object, performs various tests based on information in the command definition object, and calls the command processing program (CPP). When your program runs the DSPOBJD command and when records are written to the file, a call occurs to data management to open the file, to write the records, and to close the file.

Most of the commands in a CL program are run by using a CPP. A few commands, such as the Change Variable (CHGVAR) command, are run by inline instructions within the CL program.

When the RPG program is called, most of the RPG operations are performed by inline instructions within the program. However, internal calls are used to access system function, such as opening the database file and the printer file. Whenever you read a record from the database file or write to the printer file, you may be causing a call.

For simple read or write types of files, a high-level language such as RPG provides blocking support so that the number of calls is far less than one for every record. When the program ends, RPG does an internal call to close each of the files.

Therefore, the right answer to the question "How many calls are run by this simple application?" is that **many** calls are run.

Will ILE Affect the Performance of Calls to IBM Functions?

The answer to the question of whether ILE affects the performance of calls to IBM functions is both yes and no. At this point, ILE addresses neither the data management calls nor the calls caused when a command like DSPOBJD is run. In fact, most system functions that are called are not addressed by ILE. These system function calls represent the vast majority of calls that occur in a simple application like this one.

However, ILE programs can use some of the APIs that are provided by the system and that allow a faster call. If your application can use one of them, your system performance can be improved. In this simple application, the APIs that operate in this manner are not applicable.

In the future, IBM functions like data management may become ILE service programs and be accessed more quickly than they are today. In Version 3 Release 1, most of the IBM functions are accessed by using a traditional dynamic call.

Could ILE Be Used to Improve the Performance of This Program?

ILE performance depends on the number of times you run a CALL command from CL or a CALL operation from RPG (or your high-level language). The application makes an external call only once, when the beginning program (PRTPGMC) starts to run. At the present time, it is unlikely that ILE could be used to improve the performance of this small, insignificant test program.

Although ILE may reduce system time by microseconds, a single use of this application would not improve system performance. The type of system you use might have an effect, but you would probably have to run this program several thousand times in an ILE environment in order to save one second of operating system time.

If you focus only on call performance in this application, you are unlikely to see a significant performance gain due to ILE in this release. This may also be true for many of your existing applications that do not perform a large number of external calls. A significant payoff requires a lot of calls.

On the other hand, some applications perform better in ILE because the programs are optimized for ILE when the code is created. In the future, bringing system functions, such as opening a file, into ILE may provide better overall performance.

Chapter 9. Integrated Language Environment (ILE)

With the new features provided by ILE come additional concepts to understand and use. To make the example application in Chapters 2 through 7 as clear as possible, ILE concepts are introduced where they relate to the steps in the example. To focus on ILE itself rather than on the example, this chapter presents several ILE-related discussions.

What Is the Difference between a Module and a Procedure?

The terms *module* and *procedure* are virtually synonymous from an RPG and a CL perspective. One source member for CL or RPG creates one module, which is called a procedure when it runs. Other languages may support modules that have multiple procedures. Therefore, the terminology used in ILE is:

- You create or change modules.
- You call or run procedures.

What Does the Last Step of the Compile Process Really Do?

The last step of the Create Bound RPG Program (CRTBNDRPG) command uses a module object to create a program object. The compiler also generates the program entry procedure (PEP), which is given initial control when the program is called. The PEP then passes control to the called procedure in this module (see Figure 9-1).

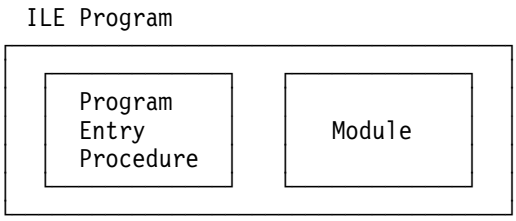

Figure 9-1. Program Object with Single Module and Program Entry Procedure

If you use the CALL command to do a dynamic call, the object you call has to have a program entry procedure (PEP).

OPM programs have a kind of PEP. It is created as part of every program. In ILE, every RPG module has a PEP; in a multiple-module program, only the PEP that is in the specified entry module is kept. The binder ignores the PEPs for all of the other modules.

The purpose of a PEP is to identify the module that is given control. This module is known as the entry module. With the CRTBNDRPG command, you create a program with a single module. The system automatically makes your only module the entry module. When you create a program with multiple modules, you need to identify one as the entry module.

ILE Program Size Compared with OPM Program Size

Table 9-1 shows the sizes of the OPM and ILE versions of RPG program PRTPGMR. Note that the debug information is significantly expanded (one reason is that it contains the compiler listing), so you should expect it to be larger. The OPM option to remove debug and the program template are identical. This is because the Change Program (CHGPGM) command supports only a single option.

Table 9-1. Comparison of ILE and OPM RPG Program (PRTPGMR) Size, in Bytes. This table compares the sizes of ILE and OPM programs under varying conditions.

Condition	OPM	ILE
With observability	33 280	104 960
Without debug information	9 728	71 680
Without program template	9 728	55 904
Without debug information or the program template	9 728	22 528

What Happens to the Call Stack?

With the program as you created it, there would be some changes to the call stack. A procedure within the module runs when it is activated.

When an ILE program is called, the program entry procedure (PEP) gets control. The PEP is part of the entry module. It calls the procedure in the entry module. The PEP name appears in the call stack. The module name does not appear as a call stack entry. Instead, it is displayed next to the procedure that is a call stack entry. If you display the call stack while the RPG program is running, you see:

```
_CL_PEP                  (CL PEP)
PRTPGMC2                 (CL Program)
_QRNP_PEP_PRTPGMR2_IL    (RPG PEP)
PRTPGMR2                 (RPG Program)
```

Are There Any Changes to Command Definitions?

The way the Transfer Control (TFRCTL) and Call Procedure (CALLPRC) commands are restricted to ILE CL programs is by adding options to the ALLOW parameter on the Create Command (CRTCMD) command. In addition to *BPGM and *IPGM, two new values are now supported: *BMOD and *IMOD. The default for ALLOW is *ALL, which is the way most commands are created.

However, if you changed some of the IBM commands for this parameter, or if you used a specific list for your own commands, you should review these commands to see if the new values should be added.

If a command written by one of your programmers returns a variable and if the command-processing program (CPP) is a CL program, these programs are probably already restricted to *IPGM and *BPGM. If you convert these CL programs to ILE type (CLLE), you should probably specify ALLOW(*BPGM *IPGM *BMOD *IMOD).

Shipping Program Objects to Other Systems

If you ship program objects to other systems, they do not need the module objects. You only need the module objects on the system on which the programs are created. If you need to change a module in an existing program on another system, you can just send the module and use the Update Program (UPDPGM) command (see Chapter 13, “Updating ILE Programs” on page 13-1).

Library Objects

The library now contains both the new bound program (type *PGM) and the two module objects (type *MODULE). The system does not distinguish between the program types created by either CRTBNDxxx or CRTPGM. At this point they have the same properties, and you perform operations on them in the same way. For example, generic functions such as EDTOBJAUT, SAVOBJ, and MOVOBJ all work on *PGM objects regardless of how they were created.

The attribute of the modules is RPGLE. the attribute of the bound program whose entry module is RPGLE is also RPGLE. For program objects, the attribute of the program is the attribute of the module that contains the PEP. For example, CLE is the attribute if the following are true:

- One C module and 200 RPG modules are bound
- The C module has a PEP
- The value on the ENTMOD parameter specified the C module

For an ILE service program, if all modules that make up the service program have the same attribute (all C modules or all RPG modules), the resulting service program has the attribute. If a mixed-language set of modules make up the service program, no language attribute is associated with the service program.

There is no way of looking at the DSPOBJD information (either on a display or in an output file read by your program) and determining whether the program has more than one module. To determine this, you can use the Display Program (DSPPGM) command as follows:

```
DSPPGM   PRTPGMI
```

DSPPGM shows you:

- The name of the module that contained the PEP when the object was created
- A list of all the modules
- The service program information of the modules that were included
- An indication of the service programs to which the modules are bound

At this point, the user of the application can still make an error by calling the bound program directly, rather than by calling the CL program. Try making this error by typing the following:

```
CALL   PGM(PRTPGMI) PARM(PRTLIB)
```

You should see the inquiry message.

What Has Been Accomplished?

This example eliminates the major deficiency of using a more modular design on the AS/400. That deficiency is the cost of the repetitive call to a subfunction. The CALLB operation is very efficient because the modules are bound together by the CRTPGM command.

Will the Bound Call Run Faster than an RPG Subroutine?

Bound calls do not run faster than an RPG subroutine. The RPG subroutine approach still provides the best performance. The point of the bound program is that the overhead involved in using a more modular design is more acceptable for highly repetitive use. The next chapter presents performance guidelines about when and when **not** to use each solution.

In this application, the date conversion is used only once per record read. The PRTLIB library is small and has only a small number of program and module objects. If the only library for which you use this program is PRTLIB or one like PRTLIB, you could use a dynamic call and the overall performance of the application would be about the same. You will not see any differences in any of these solutions when you have only a small number of calls.

What Naming Convention Should You Use for CRTPGM Objects?

In the example, the unique name of PRTPGMI was used for the program object. This requires that you decide on a naming convention. You may want to include a unique character, or characters, that tells you it is a multiple module program. Disadvantages of this approach are the following:

- It is difficult to find a convention that you can follow for most cases
- Because every name is unique, you end up with many names

The other alternative is to use the same name as the PEP module. Both the program and the module can exist in the same library because they are different object types. The disadvantage of this approach is that you cannot tell what the object is by just using the name.

Debugging

In the steps named, you took the default. Thus, the debugging information for each of the modules is in two places:

- In the module object
- In the program object (one set of debugging information for each module)

Therefore, you can debug any of the modules in the PRTPGMI2 program with the debug function. When you start the debug function, the program entry procedure (PEP) is the assumed default.

If you start using the same module in many different programs, the space requirements for taking the debug defaults can be significant. After testing the DATCVT2 function, you should consider removing the debug capability to reduce the size.

Once you create the program, you cannot remove the debug information for a single module. The only function supported by the Change Program (CHGPGM)

command is the removal of the debug information for all modules. To remove the debug information from one module, you have to use the Change Module (CHGMOD) command. Then, you can use the CRTPGM command again. In addition, you can use the UPDPGM command, specify a module that either does or does not have debug data, and update the program accordingly.

Call Stack

In ILE, the call stack is different from the call stack in OPM. The program entry procedure (PEP) for an ILE program always appears in the stack. The call stack contains the name of the program. Then, the names of any procedures that are in the stack appear.

If the system runs an instruction in the date conversion procedure, the call stack for your functions looks like this:

```
_CL_PEP
PRTPGMC5
PRTPGMR4
DATCVT2
```

When the date-conversion procedure returns, it is removed from the stack. Because LR is off, the program is still active, but it is no longer in the stack.

Using the DATCVT2 Module

- The DATCVT2 module is typical of the types of function used in many applications. It is not unique to this application.

  A better approach could be to make the date conversion procedure a service program. This reduces the size of the application objects and eliminates the need to bind the standardized functions into every program that needs one.

  However, the performance of a service program is not always the fastest. You do not always want to use a service program to accomplish a date conversion or similar action, even though the functional aspects of a service program look desirable.

- The programs pass parameter lists back and forth. An alternative to passing parameters is to use the new function of import or export of variable names. Both methods have advantages and disadvantages.

  When you use a dynamic call, you must pass a parameter list. There are other alternatives, such as using the local data area, but generally speaking you have to adhere to the strict rules for passing parameter lists.

  If you use a service program, you can pass variables by import/export or by using a parameter list. Once again, you might not choose a service program because of the performance implications.

- The CRTPGM command requires parameters for which there are no defaults. This means that you have to specify the right information each time you create an object again. Appendix A, “Packaging CRTPGM and CRTSRVPGM” on page A-1 describes some alternatives that you should consider.

Similarity to OPM

Sometimes, you must change a function that you use from multiple application programs. How you find all of the places where the function is used in ILE is similar to how you find them in OPM.

- If you copied the source code into each of the programs, you have to find the programs, change each of them, and then create them again. To help find each program that uses copied source, some programmers put a unique comment into each of those programs and scan for the comments.
- If you use an include, such as the /COPY function in RPG, you do not have to change the programs that use the function, but you do have to find them and create them again. A scan of the source should assist you here.
- If you made a dynamic call to a subprogram, you probably do not have to do anything. In most cases, the interface does not change. This is an advantage of dynamic calls.
- If you make a bound call to a procedure in the same program, you have to find the programs and create them again or use the Update Program (UPDPGM) command. These problems, along with some tips, are discussed in detail in Appendix B, "Where-Used Capability" on page B-1.
- When one of the modules changes, you do not have to re-create an entire program. You can use the UPDPGM command, which allows you to replace one or more modules in an existing program.

How Many Modules Should You Have in One Program?

The DSPPGM command shows you the number of modules currently bound into a single program. It also provides information about the maximum number of modules that can be bound. However, assuming no debug data is present, the maximum number of modules that can be bound together is 16 381.

This is potentially more than you want to handle. You need to decide on your own practical limit. Regardless of how modular your program designs become, by using CRTPGM you should only have to re-create about the same number of programs as you do by using CRTxxxPGM in OPM. You may have more CRTxxxMOD commands than you would with a CRTxxxPGM approach, but module creation should be faster because the modules are smaller. Running the CRTPGM command should be considerably faster than using CRTxxxPGM.

How Big a Program Can You Have?

Once again, the Display Program (DSPPGM) command shows you how large a single program can be. However, a program can be roughly 256GB in size.

Imports and Exports

Imports and exports are an alternative to a parameter list. The picture of the application does not change. Variables could have been imported and exported when the two RPG modules were bound together. An important restriction with this approach is that CL programs do not support the importing and exporting of variables. The CL restriction does not really change the program design. A single

program can use a parameter list to pass data between some procedures and use import and export of variables between other procedures.

When you pass a parameter list, the calling program or procedure has the data stored inside its work area. The program or procedure that you call uses an internal address to access the data from the calling program or procedure's work area. For a simple use of import and export variables, each variable that is needed should be defined as an export in one module and as an import in another.

Location of Import and Export Storage

When you define a field as an export, you are stating that the storage for the field resides in the work area of the same module. When you define the field as an import, you are stating that the storage resides in a different module. See Figure 9-2.

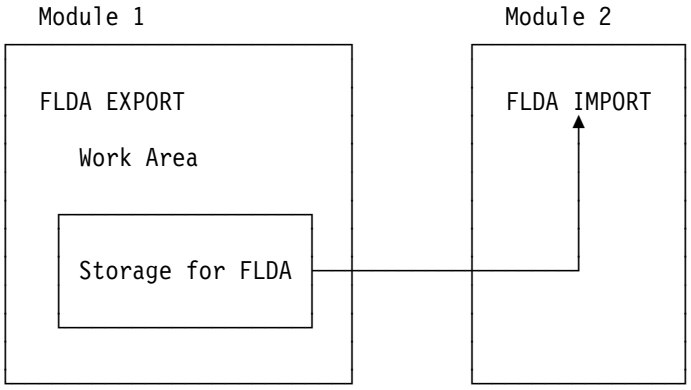

Figure 9-2. Multiple Module Program (Logical View)

In our sample application, a parameter list of three fields was passed (MMDDYY, YYMMDD, SETLR). How would you decide which field is an export of the main-line module? For example, the SETLR field is used by the subprogram, but it is the main-line program that determines how to set it. Does that make the SETLR field an export from the subprogram? Would the function work if all three fields were defined as exports from the subprogram? Would it work if all three fields were defined as imports?

From a functional viewpoint, the sample application can work with either approach. You could have had just exports from the subprogram, just imports, or a combination. There are some differences described later that may influence your decision. In general, the term export is used to describe where the storage area is, rather than which procedure manufactures the data.

What if you had said that SETLR was an import in both modules? Both modules would create successfully, and the RPG compiler would know that the storage is elsewhere. When you use CRTPGM, the function fails because no module is defining the field as an export.

What if you had said that SETLR is an export from both modules? This is not a problem, but it could be when you use CRTPGM. The default is to ensure that unique names are being exported. This means that both modules could not specify an export of SETLR.

CRTPGM allows you to have duplicate procedure or variable names. If you are using RPG or CL, you should take the default on the OPTION parameter because you will not be able to access the duplicates.

In our sample application, unique names are not a problem. However, in a more realistic environment, you must have a good naming convention to avoid potential problems.

When you export and import variables, you do not eliminate the need to define the attributes of the data in every module using the data. The function of import is not like the concept of externally described data, when you refer to a file and then just use the variable names. You have to define the attributes of every variable you are going to use for the import and export of variables.

You are responsible for a consistent definition of the attributes of the variables, just as you are with parameter lists. If the attributes differ, you receive unpredictable results when you run the program.

What Are the Functional Implications of Which Module Says Import?

Storage is allocated in only one place locally. Unlike a parameter list, the value of a field that is defined as an import is undefined. This means that you should not use the field until you have done one of the following tasks:

- Moved a value to the field
- Called the module that describes it as an export

It is valid to move data to a field defined as an import before you call the procedure that defines it as export. Thus, the SETLR field could be an import to the main-line RPG procedure and be set in the main-line procedure. You are actually moving the data to the work area of the other procedure. Because all of the procedures are part of the same program, all of the work areas are created when the program is activated.

In most applications, how you code the rest of the module does not depend on whether you passed the data by parameter list or by import and export variables. If you pass by parameter list, you normally do not attempt to use the data in a field that is filled by a subprogram until you call the subprogram. You must be aware of the RPG restriction that only a single procedure can define a field as export.

Thus, if our sample application had another procedure that wanted to use the date-conversion procedure, it would now make a difference. If the date-conversion procedure is called from multiple procedures in the same program, the date-conversion procedure must define all three fields as export.

If you have a service program (object type *SRVPGM) made up of modules that perform service types of functions, fields should be defined as export. This allows all of the procedures to use the service function. This guideline for a multiple-module program makes sense when service programs are considered. This is because you may have procedures that begin in service programs. Then you may want to change your strategy and move them to be part of a multiple-module program, or the reverse.

RPG Definition of Export and Import

To define a field as export or import in RPG requires the use of the new definition (D) specification. It requires the use of the EXPORT or the IMPORT keyword on the D specification. For example, the main-line module might add the following:

```
MMDDYY              EXPORT
YYMMDD              IMPORT
SETLR               EXPORT
```

For the sample application, the main-line RPG module requires that all the fields be import types.

You have to define the attributes of each field even though the data may reside in a different procedure. This same rule exists for parameter lists. As with parameter lists, you need to be very consistent about the attributes you give to each variable. If you are not, the results can be unpredictable and can cost you extra debugging time. There is no check when CRTPGM is used to determine if you have been consistent.

Is Any Binder Language Needed?

Binder language cannot be used to create a program object. However, binder language can be used to create service programs (*SRVPGM). The CRTPGM command defaults assume that both of the following are available to any other procedure within the same program:

- All of the procedure names
- All of the variables defined as export

Variables that are **not** defined as export or passed as parameters cannot be accessed by another procedure. For example, the WORK2 field used in the date conversion procedure cannot be accessed by the main-line procedure to read or change the field. Variables that are not defined as export or import are known only to the procedure in which they are defined. Both procedures could define a WORK2 field, and the definitions could differ.

Is Any Binding Directory Needed?

There are parameters on the CRTPGM and CRTBNDRPG commands that relate to a binding directory, but they are not needed for this simple application. Use CRTPGM to create a multiple-module program from which one of the procedures does a CALLB to a procedure that does not yet exist.

A binding directory can specify *MODULE and *SRVPGM objects. A CALLB is a call to a procedure within a module object. The binder looks for an exported procedure by that name and, if it is a module, binds the module by copy. If a service program provides the procedure, a bind by reference occurs.

Using a Parameter List or Using Import and Export of Variables

When you work with multiple-module programs, you can pass a parameter list or you can import and export variables. The things to remember are that:

- RPG allows additional control when using parameters through Factor 1 and Factor 2 move capability on the PARM statement.
- Exported fields are not reinitialized after LR processing or abnormal ending. They stay completely under user control until the activation group is deleted.
- CL allows you to pass only a parameter list.
- You can reduce the overhead of passing parameters by passing a single parameter that is really a data structure. If you pass a single parameter (for example, a data structure), there is essentially no performance difference between the following:
  - Passing a parameter
  - Importing and exporting variables
- Either solution requires that the variables be defined in each module that specifies them.
- You can avoid the default restriction (allowing only single modules to export unique names in the same program), by using a naming convention.

Chapter 10. Modularity

Although redesign is unnecessary, it may make sense to create more modular programs. Because increased modularity in program design can enhance benefits of the new ILE features, it is discussed separately in this chapter.

For some of you, modular programming may represent a new direction. However, some of you may feel that your applications are already modular.

In fact, it could be argued that the sample application already is modular (multiple parts exist, and each part performs a specific function). It could also be argued that the sample application is not very close to true modular programming.

Many RPG programmers write large programs that do not call subprograms. Such programs generally perform well in OPM.

Degrees of Modular Design

It is obvious that there are degrees of modular design. As depicted in Figure 10-1, the continuum ranges from barely modular all the way to more modular than is useful. The traditional linear approach to the left is modular only in the sense that some of the lines of code are organized by task. The other extreme on the right is represented by many small modules, each with perhaps only one or two lines of code. This extreme could make maintenance very difficult.

We envision the central point of modularity on the continuum as offering you the most potential to get the best from ILE. The figure suggests that, when it makes sense, a reasonable amount of code is isolated in a reasonable number of modules so that they are convenient for reuse, they compile quickly, and they are easy to track.

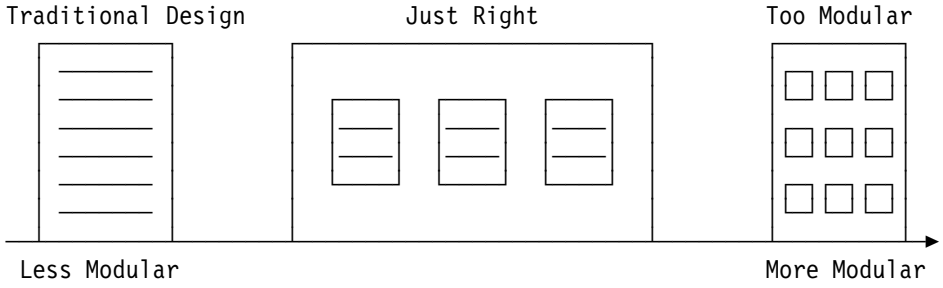

Figure 10-1. Degrees of Modular Design

Traditional OPM Design

The traditional OPM design is not very far along the line of modular possibility. However, traditional designs can be adjusted to make them more modular.

Just Right

Just right on the modularity continuum covers a variety of modularity levels. There is no definitive level of modularity. It is defined by an appropriate combination of factors, such as:

Design
Performance

Coding
Testing
Maintenance
Growth

Two equally good programmers might easily disagree on what just right means for a given application, and yet both could achieve effective results.

Too Far

There is also a point on the continuum that represents going too far because there are too many modular pieces. It is also not logical to have modules that are too small (for example, containing only one line of code).

Modular Programming Debate and Trends

The debate about what is the best approach to programming will go on as long as there are programmers. There are no clear winners in the modularity debate. Instead, there are alternatives. Some programming languages lend themselves better to a modular approach.

ILE provides more benefit in your business if you work toward more modular application design. Although there is good AS/400 system support, it may take a little time to create applications with just the right level of modular design. Fortunately, it is not necessary to change anything at the very moment the ILE compilers are delivered. You can gradually adjust your application design to the appropriate level of modularity.

As with most programming concepts, you need to work with modularity before you can decide the best use of this approach in your environment. The question to ask is whether it is worth the initial investment to code for reuse versus getting an application working as fast as possible.

If you are new to modular programming, the payoff is more likely to occur in a future application. You may need to gain some experience with modular programming before deciding what is the right level to use for your application requirements. In addition, you need to consider issues of maintenance, service, and productivity. For most people, this means that you start out simple, you learn from your initial efforts, and you find out what works best.

The general AS/400 programming community is moving toward more modular programming. The system has the function to support varying degrees of modular design. You need to make an investment to learn the concepts and to practice the new approaches. How far you go is your choice.

Advantages of Modular Programming

An advantage of ILE is that the AS/400 system is now capable of supporting a greater degree of modularity. There is now function and performance that makes excellent use of a greater degree of modularity than was practical in the traditional AS/400 application approach.

Benefits of Modular Programming

There are numerous benefits to modular programming.

Modules can be reused.

Reuse of modular code in multiple applications is an obvious way to enhance programmer productivity. Writing commonly used routines in external modules decreases the need for repetitive coding efforts. It also allows a proven, optimized routine to be used effectively in many different programs.

Libraries of code (including purchased modules) can be collected and used at will in a variety of applications. You can select packages of routines and blend them with your own programs.

With ILE, reuse has an additional dimension. Programmers can write in the languages they choose, and routines in any ILE language can be used by all AS/400 ILE compiler users.

Maintenance is simpler.

Modularity contributes to simplified maintenance because it isolates the changes. When maintenance is required, the areas to be changed do not need to be searched for in large amounts of linear code. Rather, the process can be approached by checking individual modules and making the necessary changes within them. There is obvious value in having future changes isolated in a single module rather than spread throughout an entire program.

Testing is also simplified.

The gains in testing code also have to do with isolating the changes. Individual modules can be tested and then changed if necessary. On large projects, certain modules can be tested and verified prior to completion of the entire program and the test of the entire system.

Compile times are faster.

Smaller pieces of code compile faster. The individual modules that make up a larger application are smaller, so compile time is less. Only the tested (or changed) modules need to be compiled again. This decrease in recompiling increases productivity.

Programming resource can be used more effectively.

Modular coding techniques allow for more effective distribution of the work load among developers. More programmers can work simultaneously on multiple modules that make up a single function. Also, individual programmers can work on the portions of code best suited to their experience and skill level.

Migration from other platforms may be easier.

Again, because the code is located in modules, it is in smaller and more isolated pieces. This isolation can contribute directly to the ease of migration. Also, modular programs from other platforms, such as UNIX**, can be migrated to the AS/400 system and incorporated into ILE programs.

Modular programming is the basic underpinning of object-oriented programming.

Designing applications in a modular fashion is a necessary stepping stone toward object-oriented programming.

More Modularity with ILE

Using a greater degree of modular programming means writing smaller programs and making repeated use of existing standard functions. These standard functions include:

Functions you have written
Functions IBM has supplied
Functions other vendors or users have supplied

Even in ILE, not all modular applications provide every benefit mentioned. The reason often involves both the application problem to be solved and the design used.

You may have used modular design to a significant degree on your AS/400 system prior to ILE. The major additional benefits of modular programming with ILE are:

- Performance of bound programs
- Use of service programs
- Improved run-time control (for example, activation groups)

You may be interested in converting to a more modular design in order to:

- Achieve the potential advantages of application maintenance.
- Maximize the potential of using existing code in new applications. It is most likely that you will use more modular design for new applications rather than spend time converting old ones.

It is important to realize that more modular design is not necessarily the best solution to every application design issue. You have to decide how far you want to go.

Possible Disadvantages of Modular Programming

Although there are many advantages to modular programming, we must also acknowledge that there are occasional tradeoffs to be considered. These tradeoffs might include the need to consider that a more modular design may exaggerate or compound a problem. The problem itself may not be limited to the modularity of design, although the problem may be affected by it. Therefore, you may want to consider the effect that increased modularity might have in some of the following situations.

The application has numerous parts.
When there are a lot of parts, there is a greater chance that parts may be missing or down level. This risk can be minimized by having tools that provide where-used information and automatic re-creation functions.

Passing parameters or export and import variables can be error prone.
Using externally described data is usually a better approach. Extra debugging time occurs unless the values have identical attributes in all modules.

One part has information that another part needs.
When a program is split into multiple parts, one part may have information needed by another part. Therefore, it is common to make additions to the parameter list that is passed. So, while a single program contains all the information, multiple-module programs may require extra maintenance.

You may not be able to see all the code or refer to all the documentation.
It is easier to understand a program if you see all the code rather than reading documentation about multiple subprograms.

Modular programming can create a need for education.
People using the modules need to know which modules exist and how to use them.

Modular programming can be overdone.
It is possible to overdo modularization. See “Degrees of Modular Design” on page 10-1 for more information.

Chapter 11. Debugging

Debugging in ILE is significantly different from debugging in OPM. In this chapter, we discuss some of the features of debugging in ILE.

New Debug Structure

In ILE, debugging is different in several ways:

- A new source debugger exists for ILE programs.
- You actually debug at a module level, rather than at a program level as in the OPM. However, the programs, not modules, are added and removed from debug.
- The debug information is physically separated from the program template.

If you used PDM option 14 to create the RPG program, you used the defaults on the CRTBNDRPG command. This means that the observability information was included in the program. The observability information is now split into two parts:

Debug information
Program template

Both pieces of the observability information are associated with the module and not with the program (see Figure 11-1).

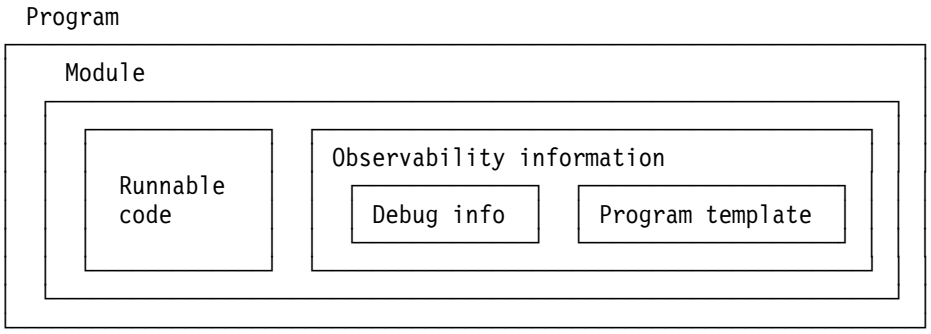

Figure 11-1. Debug Structure in Program Module

If you used the CRTBNDRPG command to create the program, you can get rid of one or both pieces of the observability by using the Change Program (CHGPGM) command. The Remove observability (RMVOBS) parameter on the CHGPGM command now supports several options, such as:

*CRTDTA	Deletes just the program template
*DBGDTA	Deletes just the debug information
*ALL	Deletes both the program template and the debug information

The compress object (CPROBJ) parameter works on an object basis. It was enhanced to support *MODULE and *SRVPGM objects. Thus, no corresponding change was required to CPROBJ for these new options.

If you bind several modules into one program, the CHGPGM command operates on all of the modules. If you request RMVOBS(*DBGDTA), none of the modules in the

program can be debugged. If you have just created a module, you can use the Change Module (CHGMOD) command to remove observability.

New Source Debugger

As part of ILE, the debug function on the system is significantly enhanced. The new source-level debugger can be used only with ILE programs.

When you use the Start Debug (STRDBG) command and name a program, the system determines the program type (OPM or ILE). For OPM programs, the same function exists as prior to ILE. You still use the Add Breakpoint (ADDBKP), Add Trace (ADDTRC), and Change Debug Variable (CHGDBGVAR) commands.

For ILE programs, the interface changes so that you are presented with an interactive display. The important thing to remember is that the system decides which interface to use based on the type of program (see Figure 11-2).

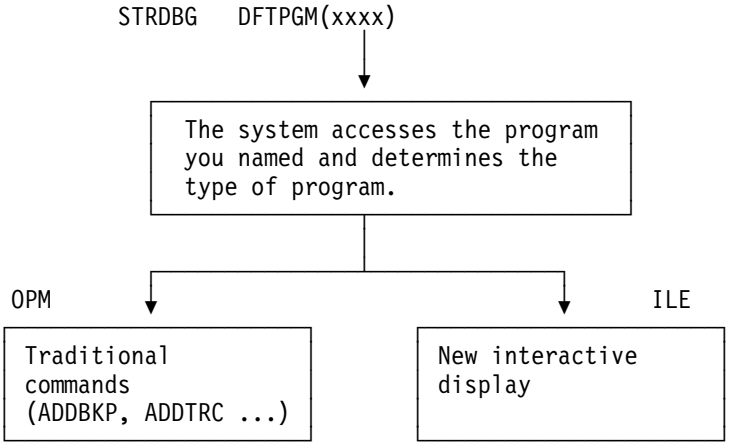

Figure 11-2. New Interactive Display As Part of ILE Debug Function

What About Debugging ILE CL Programs?

You also use the source debugger to debug ILE CL programs. The process is the same as debugging an ILE RPG program. The Start Debug (STRDBG) command determines the type of program (OPM or ILE). Then, you use either the existing commands (for example, ADDBKP) for OPM-type programs or the new source debugger for ILE-type programs.

Program Views

The new interactive display allows you to see several different views of the program. The default is the source view, as long as it exists. If only the listing view exists, then you get the listing view. However, you choose the one you want from the following:

Source view

In the example, you used the CRTBNDRPG command, so the module became the program. If you change the value on the DBGVIEW parameter from *ALL to *STMT and specify STRDBG DFTPGM(PRTPGMR2), the new source debugger is called and you can view the source statements from the source

member PRTPGMR2. The source member must exist (no part of the debug information contains the original source statements).

Listing view
With PRTPGMR2 you see the RPG compiler listing that is stored in the debug information of the program. Storing the listing increases the function of the debug capability but increases the size of the program.

Copy view
This is the source plus any statements brought into the compilation with the /COPY statement. In the PRTPGMR2 source, no /COPY statement is used, so this is an identical view to the source view.

Statement view
The RPG default is *STMT.

You cannot eliminate just one of the views. If you remove debug data, all three are eliminated.

If a program contains multiple modules, the entry module displays first. A program created with the CRTBNDRPG command has a single module, which is also the entry module. If a program contains multiple modules, all the modules with debug data of a program are in debug if the program is in debug. If you use F14, you select the particular module source you want to currently view. You can also debug multiple modules at the same time in a manner similar to that of debugging multiple programs with the OPM debug function.

Each of the debug views has a command line devoted to debug functions. It cannot be used for entering system commands. However, you can use F21 to access a system command entry display. On the debug command line, you enter commands to add breakpoints, to display variables, or to change variables.

The differences in debug capabilities between ILE and OPM are:

- ILE debug adds a step capability similar to that provided by the trace function of OPM debug. Tracing program flow is done with the STEP debug command.
- ILE debug allows debugging of one module, multiple modules, and multiple programs.

Debug Example

Assume you want to debug PRTPGMR2. On a command entry display, enter:

```
STRDBG    DFTPGM(PRTPGMR2)
```

The system determines that PRTPGMR2 is an ILE program and uses the new debugging support. The debug function determines the entry module of your program. The source view is shown on the interactive debug display.

Breakpoints can be either simple or conditional, and they can be entered several ways. Assume you want to set a breakpoint at a particular line. The debug display supports a command line for debug functions. To enter system commands, you use the F21 function key. On the debug command line, enter Break and the line number of the desired stop location. After entering the Add Breakpoint (ADDBKP) command, you call the CL program:

```
CALL   PGM(PRTPGMC2)
```

When the RPG program is called, the system recognizes that debug mode is being used and displays the same view that you saw before.

To display the value of the field XXXXX, enter the following on the debug command line.

```
EVAL XXXXX
```

There is significant additional function associated with the new debugger. For more information, see the *ILE RPG/400 Programmer's Guide*.

Chapter 12. Activation Groups

Creating an activation group is a way to place walls around an application function. Do this to minimize interference with other application functions in the same job.

The walls set up by an activation group are not very high, and there are many situations in which one activation group can affect what is happening in another activation group. For example, the Change Job (CHGJOB) command changes the attributes for the entire job and could have an effect on what happens in a different activation group.

An activation group always refers to the same function within the same job. Each job can have more than one activation group, and activation groups are not shared across jobs. If you have group jobs or system request jobs, each job is considered separate and has its own activation groups.

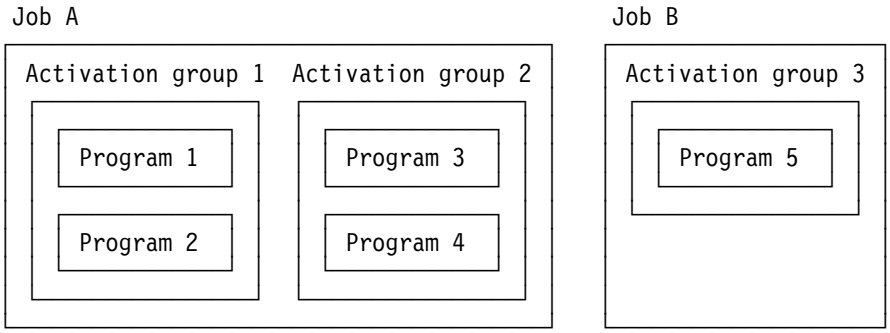

Figure 12-1. Activation Groups within Separate Jobs

Here are some examples of uses for application groups.

- Two application functions want to open the same file as shared, but they want a separate cursor. For example, they both want to control the position from which the next record in the file will be read, or they need different open options on the open data path (ODP).

  With OPM, when you open the same file that is shared, all programs that use the file share the same cursor. This means that an operation code like READ, which reads the next record in the file, may not read the next record based on the last read from the same program. The open options of the ODP are set by the first open. Another program opening the same file that is shared cannot ask for additional options. By opening the file shared in a separate activation group, each application function can have a separate cursor.

- Assume that you want to use commitment control in your application. After you have performed some uncommitted changes, call a subfunction or a function like OS/400 Office. The subfunction also wants to use commitment control. When the subfunction does a COMMIT in OPM, your changes are also committed because there is only one commitment control block for the job. With ILE, each activation group has its own commitment control block and can operate independently.

- Using a service program in a separate activation group. This can be a significant performance advantage because you can activate the service program once for the duration of the entire job or application.

How Activation Groups Fit with Other Job Concepts

An activation group is a piece of a job, but should not be thought of as a different job. For any given job, there is only one:

Library list
QTEMP
Job log
Set of job attributes (changeable by CHGJOB)
Call stack
Set of locks
Process access group (PAG)

Overrides are a special case (see the *ILE Concepts* manual.)

Running in a separate activation group cannot protect an application function completely. For example, an application developed to run in a separate activation group cannot change a function like the library list without potentially affecting the rest of the job.

Default Activation Groups

The system supplies two default activation groups for each job when it is started.

System Activation Group
Most system functions run in the system activation group, and it is of no concern in your design.

Default Activation Group
The default activation group is the place where all OPM programs run. OPM programs do not have a choice about where they run.

ILE programs can run in the default activation group if either of the following is true:

- ACTGRP(*CALLER) is specified on the create command, and the program is called from a program already in the default activation group
- DFTACTGRP(*YES) is specified on the CRTBNDRPG command

If you specify DFTACTGRP(*YES) on the CRTBNDRPG command, the resulting program behaves like an OPM program in the areas of scoping open data paths, scoping overrides, and using RCLRSC. This high degree of compatibility is due in part to its running in the same activation group as OPM programs, namely, in the default activation group.

However, with this high compatibility comes the inability to have static binding. Static binding refers to the ability to call procedures (in other modules or service programs) and to use procedure pointers. In other words, if you specify DFTACTGRP(*YES) on CRTBNDRPG, you cannot use the CALLB operation in your source. Nor can you bind to other modules during program creation. Similarly, you cannot use CALLPRC in a module created by CRTBNDCL with the default value DFTACTGRP(*YES).

If you specify DFTACTGRP(*NO) on the CRTBNDRPG command, the resulting program has ILE characteristics such as static binding. At program-creation time, you can specify the activation group the program is to run in and any modules for static binding. You can call a service program or any other procedure if you specify

a binding directory on the BNDDIR parameter. In addition, you can use CALLB in your source.

Major Difference between OPM and ILE Programs in an Activation Group

The major difference between system handling of OPM and ILE programs is what happens to the working storage when the program ends. For RPG, this means when the program ends with LR on.

- For OPM programs, the working storage is logically deleted. The space can be reused by the next program. If you call the same program again, the working storage must be reassigned.
- For ILE programs, the working storage remains. If RPG ends with LR on, the working storage remains assigned but is initialized the next time the program is called. ILE CL programs operate like RPG programs with LR on.

Assume that your application calls program PGMA, which calls PGMB, which calls PGMC. Also assume that all programs are running in the same activation group and that all programs are RPG programs that specify LR when they return. When PGMC is running, the call stack looks like this:

```
PGMA
PGMB
PGMC
```

Each program has working storage assigned to it. When the program finally returns to PGMA, PGMB and PGMC are no longer in the call stack. The working storage for the job would logically appear as it does in Figure 12-2.

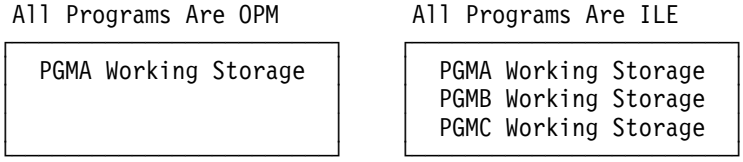

Figure 12-2. Working Storage in OPM and ILE

If PGMA calls PGMB again, the OPM solution must assign the working storage for the program. In ILE the assignments are retained, and the program only has to initialize the working storage before it begins.

This difference occurs even if the ILE program is running in the default activation group.

Assigning a Program to an Activation Group

There is no command to start an activation group. A program is assigned to an existing activation group based on what is specified for the ACTGRP parameter, which exists on several commands. An **unnamed activation group** is started when you call an ILE program that was created as ACTGRP(*NEW). This is the default on CRTPGM, but you should avoid taking the default unless you understand the implications.

Starting a new activation group involves some overhead. From a performance viewpoint, you are usually better off to specify *CALLER (run in the same group) or to provide a name.

When you specify a name for ACTGRP, the program is assigned to a **named activation group.** If the activation group does not exist, it is automatically created.

Activation groups end in different ways, depending on their type. The two system activation groups can never be ended by a command or function. They end automatically when the job ends.

Assume that you call a series of programs and that the flow is as depicted in Table 12-1. The table shows how each program was specified for the ACTGRP parameter.

Table 12-1. Program Calls with ACTGRP Parameters. This table shows the way each program was specified for the ACTGRP parameter.

Program	Environment	Parameter
PGMA - Calls PGMB	ILE	ACTGRP(*NEW)
PGMB - Calls PGMC	ILE	ACTGRP(*CALLER)
PGMC - Calls PGMD	ILE	ACTGRP(GRP1)
PGMD - Calls PGME	ILE	ACTGRP(*CALLER)
PGME - Calls PGMF	OPM	
PGMF - Calls PGMG	ILE	ACTGRP(GRP2)
PGMG	ILE	ACTGRP(GRP1)
Uses service pgm SRV1		
Uses service pgm SRV2		
Uses service pgm SRV3		
SRV1	ILE	ACTGRP(*CALLER)
SRV2	ILE	ACTGRP(GRP2)
SRV3	ILE	ACTGRP(GRP3)

The following activation groups exist, and the programs are assigned as shown in Figure 12-3.

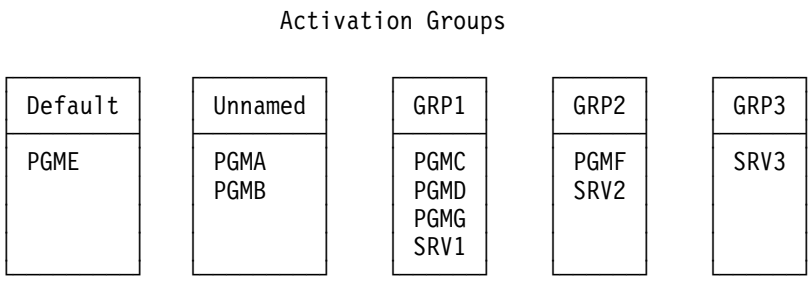

Figure 12-3. Activation Groups with Assigned Programs

The following should be noted from this example:

- If a program after PGMC specifies ACTGRP(*NEW), another unnamed activation group can start. Each use of *NEW starts a new activation group.

- Once PGMB calls a program that specifies a named group, no other program lower in the call stack can be assigned to the unnamed group. To reuse the unnamed group, you have to get back to PGMA or to PGMB and call a new program with ACTGRP(*CALLER). This is one of the reasons to specify ACTGRP(*CALLER) or to use a specific name. The default of ACTGRP(*NEW) tends to start activation groups that you do not need.
- In the middle of a series of ILE programs, you can call an OPM program such as PGME. OPM programs are always assigned to the default activation group.
- The service programs can use an existing group or a named group. If the activation group does not exist, it is created. The ACTGRP parameter on CRTSRVPGM does not support *NEW, so it cannot start an unnamed group.

Unnamed Activation Groups

An unnamed activation group is started when a program specifies ACTGRP(*NEW). It is ended when the program that caused the start of the activation group ends and is removed from the call stack.

Assume you called a series of ILE RPG programs that were specified as:

```
PGMA    ACTGRP(*NEW)
PGMB    ACTGRP(*CALLER)
PGMC    ACTGPR(*CALLER)
```

When control is returned to PGMA, the working storage still remains for each program (see Figure 12-4).

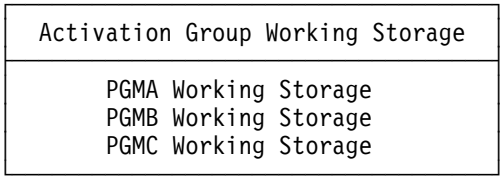

Figure 12-4. Activation Group Working Storage Remaining

When PGMA ends, the activation group is ended (deleted). All the working storage is logically deleted. If PGMA is called again, the activation group starts again and new working storage is assigned.

PGMA can be ended in a variety of ways:

- The normal method is that PGMA returns with LR on.
- If PGMA returns with LR off, the system ends the activation group and the working storage for the program is logically deleted. The next time you call PGMA, a new unnamed activation group is started with new working storage assigned. You may not get the expected results.
- PGMA ends and the activation group is ended if an escape message is sent by:
  - PGMA
  - One of the subprograms to a program higher in the call stack than PGMA

Unnamed Activation Groups after PGMA Is Ended

With programs assigned to activation groups as they are in this illustration, you can see what happens to unnamed activation groups after PGMA is ended. PGMA was the first program called and specified ACTGRP(*NEW), which caused an unnamed activation group to start.

When PGMA is ended, it is removed from the call stack and the system ends the unnamed activation group. At that point the working storage for the programs that were activated for the named activation groups still exist. This is true even though none of the programs are in the call stack. The major difference between named and unnamed activation groups is whether the working storage still exists when the program that started the activation group ends (see Figure 12-5).

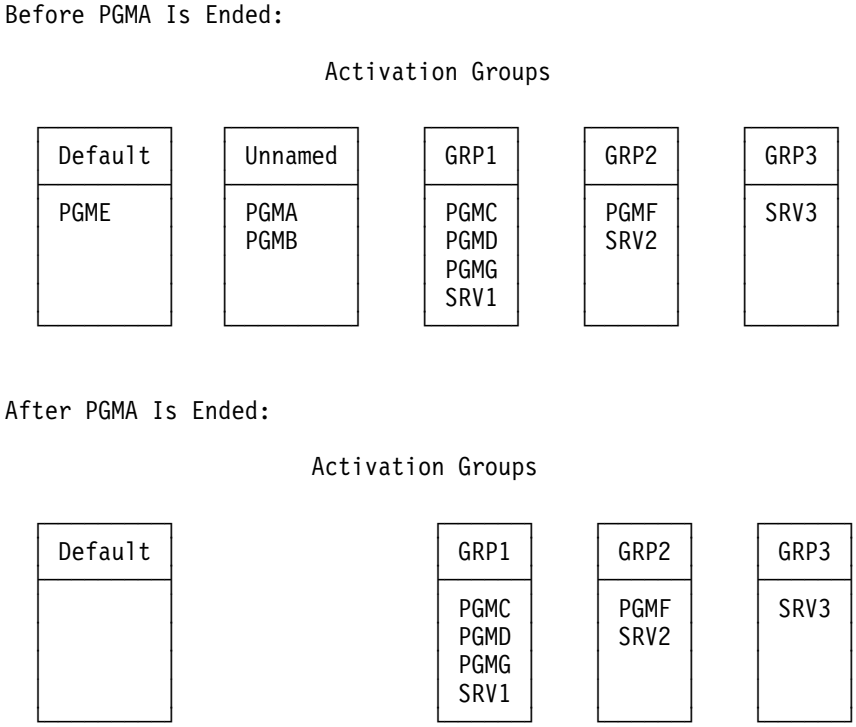

Figure 12-5. Working Storage Existing for Activation Groups. This figure compares the working storage that exists before PGMA is ended (top) and after PGMA is ended (bottom).

The unnamed activation group is automatically deleted by the system along with the working storage for both PGMA and PGMB. Until PGMA ends, the working storage for PGMB still exists in the activation group. Until the program that started the activation group ends, there is no difference in working storage for unnamed versus named activation groups. PGME is an OPM program, and the working storage is logically deleted if the program returns (assume LR is on when PGME returns).

Named Activation Groups

A named activation group is assigned for a program (or service program) that specifies a name for the ACTGRP parameter. If the activation group does not exist, it is automatically created by the system. It can be created by either a program or a service program. The major differences between named and unnamed activation groups are:

- For a named activation group, the working storage is not logically deleted for any activated programs until the entire activation group is ended. The activation group does not automatically end when the program that started the activation group is no longer in the call stack.
- A named activation group can be started by a service program.

QILE Activation Group

If DFTACTGRP(*NO) is specified, the system uses the name QILE as the default assigned for the ACTGRP parameter for CRTBNDRPG and CRTBNDCL. QILE can be considered another named activation group. It is not unique. QILE has the same properties as does any named activation group. You can use the name QILE in your applications.

Ending a Named Activation Group

All activation groups are automatically ended by the system when you end the job. You can end an activation group that you no longer need by using the Reclaim Activation Group (RCLACTGRP) command. For example, you could specify:

```
RCLACTGRP     ACTGRP(GRP1)
```

However, the system does not allow you to end an activation group if a program on the call stack is assigned to the activation group.

When starting a major new application area, you may want to place all the programs that are part of it in a named activation group. When you end the application area, you have the choice of retaining the activation group or ending it.

- By default, the activation group is retained. You could start a different activation group and then call it again later. Since the activation group still exists, you should see a faster initialization time for all previously activated functions.
- To end the activation group, you must run the RCLACTGRP command from a program that is assigned to a different activation group. You cannot run RLCACTGRP within the same activation group because the program running the command would be on the call stack. (You could also call the CEETREC API or, for abnormal ending, the CEE4ABN API. CEETREC acts as a giant LR. The activation group disappears if all programs in the activation group end.)

Here is a typical solution.

```
STARTUP                   OPM
  Calls INZAPP1
INZAPP1                   ILE          ACTGRP(GRP1)
  Calls xxx
```

The STARTUP program is an OPM program, so it automatically runs in the default activation group. You can have an initial program that is an ILE program specified as ACTGRP(*CALLER) that also runs in the default activation group.

When program INZAPP1 is called, the system starts a new activation group, GRP1. Assume that all subsequent programs (or service programs) used for the application specify either GRP1 or *CALLER. When INZAPP1 ends, it is removed from the call stack. At that point, all the working storage for any programs that were

activated in GRP1 still exist in the activation group. You can use the programs again or end the activation group.

Using RCLACTGRP to end the GRP1 activation group is valid from the STARTUP program because no program assigned to GRP1 is on the call stack. It is also valid to use RCLACTGRP from an OPM program.

The *ELIGIBLE value on the ACTGRP parameter of the RCLACTGRP command allows you to end all eligible activation groups. The eligible activation groups are those that do not have an assigned program currently on the call stack. This value never ends the two default activation groups started by the system.

What Does RCLRSC Do as Compared with RCLACTGRP?

You cannot specify RCLRSC in an ILE program. RCLRSC operates only on the default activation group because:

- You can specify RCLRSC only in an OPM program
- There is no parameter to specify an activation group

If you take the default, RCLRSC closes any files and logically deletes the working storage of any programs that are no longer on the call stack. It operates the same on both OPM programs and ILE programs.

Can the Same Program Be Run in Multiple Activation Groups?

You can run the same program in multiple activation groups. If you specify the program as ACTGRP(*CALLER), the program is assigned to the caller's activation group and multiple usage can result. This means that the program has multiple sets of working storage within the same job.

What Is a Typical Activation Group Design?

With separate application areas for user work, you should have all programs and service programs for specific areas in uniquely named activation groups. This allows you to separate any data management functions and to end the activation group, if needed.

If you have general-purpose service programs that can be used by any application area, a general-purpose activation group makes sense. If you take this approach, you cannot import and export variables because of crossing activation group boundaries. You must pass a parameter list for this approach.

As you consider which activation groups to use, keep the following in mind:

- The system has some overhead associated with each activation group. Therefore, you may want to specify only the ones you need.
- Most ILE languages have run-time routines that are not seen by the user. These run-time routines are created as service programs specified as ACTGRP(*CALLER). Thus, every time you start an activation group that runs an RPG program, the RPG run-time routines are activated in the same activation group. This is also true for procedures in service programs.

- A strategy that always uses a name for an activation group or *CALLER allows you better control.
- If the working storage gets too large, you may consider ending an activation group to reduce the effect on job performance. There is only one PAG for the job. Once you activate a program for a named activation group, the working storage remains in the PAG unless you end the activation group. You cannot end part of an activation group, so it makes sense to isolate the function in case you need to get rid of it.

To create a reasonably simple yet flexible design, you can begin with the following strategy and then consider changing it as you gain experience or have specific needs.

- For any cross-application service programs, use a specific named activation group such as GENLSRV. You must pass a parameter list to the procedures used in this type of service program.
- For all other ILE programs or service programs, use a specific activation group associated with the application area. Do not use any defaults for the ACTGRP parameter on CRTBNDRPG, CRTBNDCL, CRTPGM, and CRTSRVPGM.
- If you have cross-application programs that are not in a service program, specify them as ACTGRP(*CALLER).

Chapter 13. Updating ILE Programs

This chapter explains how to maintain existing ILE programs.

Replacing an Existing Program or Modules within the Program

If you need to replace one or more modules in an existing program, there are two solutions:

- Use the Create Program (CRTPGM) command again and name all the modules in the program.
- Use the Update Program (UPDPGM) command and name one or more modules that should be replaced in the existing program.

Update Function

Both CRTPGM and UPDPGM allow you to replace modules while active users are using the old version. The rules that determine whether this function operates correctly are the same as those for replacing an OPM program that is in use. If you change the interface to the program while it is in use, you can cause error conditions. If you are changing only the internal functions of the program, the replace function should work correctly even if there are active users of the program.

When you replace an existing program, the old version is placed in library QRPLOBJ just as it is for OPM programs. Library QRPLOBJ is cleared each time you IPL the system. If there are active users of the program when you replace it, the active users continue to operate from the version in QRPLOBJ without really knowing it. No message is sent to them.

If you replace a program with CRTPGM, you must have all the modules on the system. If you use UPDPGM, you need only the modules that are being changed.

UPDPGM Restrictions

UPDPGM has a few major restrictions:

- You cannot directly add a module to a program that does not contain the module object. But, by following these steps, it is possible to get a new module bound into the program on an update request:

  1. Change a module that was used to create the program. Have that module import a procedure or variable, and specify that module on the update command.
  2. Create another module that provides exports for the imports.
  3. Specify these modules in a binding directory, and specify the binding directory on the update request.

  Using this procedure, you add a module to the program.

- You cannot change the program entry procedure (PEP).
- When you create a program, the default is to allow the use of UPDPGM. You can prevent the use of the UPDPGM command by specifying UPDPGM(*NO) on the CRTPGM command.

If you wanted to replace RPG module PRTPGMR4 in program PRTPGMI, you would specify:

```
UPDPGM   PGM(PRTPGMI) MODULE(PRTPGMR4)
```

If you are keying in the steps as you follow the discussion, you could try this command now.

If a bound program specifies that it can be updated, you can use the CRTxxxMOD command (in this case, CRTRPGMOD) and use the resulting *MODULE object on the UPDPGM command. An advantage of the UPDPGM command is the ability to change the debug capability of a module. If you are having difficulty with a module, you can change the debug capability and replace the module to provide different debug capability.

Using the UPDPGM Command to Replace Existing Modules

If you use the Update Program (UPDPGM) command to replace existing modules, it is not necessary to keep the modules on the system. When you need to make a change, you change the source, use the CRTxxxMOD command, and use the UPDPGM command. However, if you were going to use the DATCVT2 module again in a different program, the module object must exist.

The decision you must make is whether to save space by deleting the module objects (or some of them) or to save the time it takes to create them again (and the confusion of extra steps) if you also need to create the program object again. If you have the space, there is a definite advantage in leaving the module object on the system.

Another advantage is that you can use the UPDPGM command to help in debug. For example, assume you ship the bound program and, because of space considerations, you delete the debug data. If you had a problem that needed to be debugged, you could do the following:

1. Ship a module that had debug data with it
2. Update the suspect module
3. Debug the program

Chapter 14. Service Programs

Earlier, you used ILE function to create a bound program with three modules. That program looked like Figure 14-1.

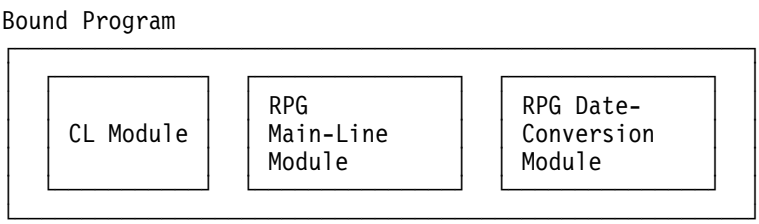

Figure 14-1. Bound Program Containing Three Modules

Although this design performs well, the use of an ILE service program can help you better manage the amount of storage. The amount of storage associated with a program of this type tends to be large because modular programming leads to a large number of utility functions. All of the physical space used by the modules is duplicated in the program object.

Binding of Service Programs

A service program is not bound to its caller until activation time. However, much of the work needed to bind the service program is done at bind time. An example is determining which modules contain the definitions of exports needed by the calling program and resolving these definitions. This approach looks like Figure 14-2.

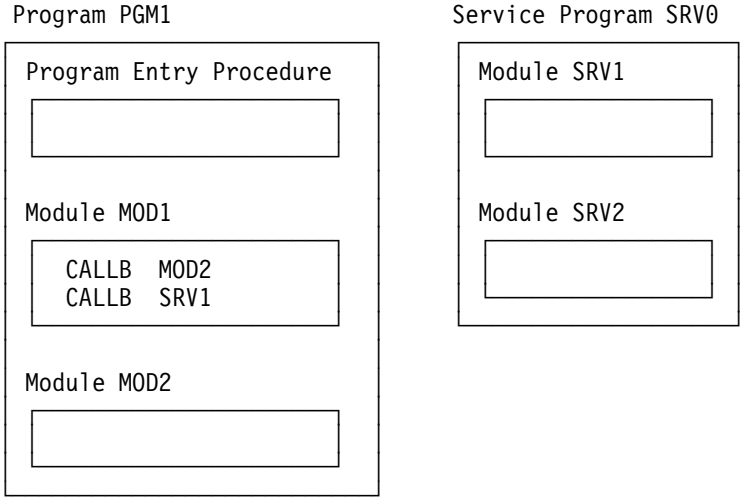

Figure 14-2. ILE Program and Service Program

Program PGM1 is created from two modules, MOD1 and MOD2. The runnable instructions in both modules exist within the program. These modules are said to be **bound by copy**.

Program PGM1 uses service program SRV0, but the runnable instructions in SRV0 do not exist in PGM1. SRV0 is therefore said to be **bound by reference** to PGM1.

Service program SRV0 does not have a program entry procedure (PEP). Therefore, you cannot use a dynamic call to access SRV0. You cannot use the CALLB to call a service program either. CALLB can be used only to call procedures within service programs.

To create PGM1, you use the Create Program (CRTPGM) command. On the MODULE parameter, you specify modules MOD1 and MOD2. On the BNDSRVPGM parameter, you specify SRV0 as a service program needed by PGM1. The system performs some binding to SRV0 at bind time, but does not copy any runnable code from SRV0 into PGM1.

If the service program is listed on the CRTPGM request, the system matches the export of the service program with the import of the modules to be bound by copy. It also does binding.

When PGM1 is called, the system determines which service programs it needs to activate and whether they are already activated. It then binds the program and the service program together.

The binder has specified which service programs are needed at activation time. Activation goes looking for the service programs that are needed. The final connection is made at activation time. The binder provided the initial connection at CRTPGM time.

The binding occurs when you call PGM1, rather than when the CALLB is actually used. When CALLB is used, the program is already bound, and at that point it performs as fast as if the service program were part of PGM1.

Because of the binding that occurs to the service program when CRTPGM is run, the service program must exist before you use CRTPGM. A service program is created by using the Create Service Program (CRTSRVPGM) command.

In Figure 14-2 on page 14-1, there are two procedures in the service program. Service programs normally contain multiple procedures and data.

Additional Service Program Information

A service program is a middle ground between a tightly bound program and a program that you call with a dynamic call. You might decide that any procedures that are called in more than one program should be part of a service program. This decision favors maintenance of your application, but it may not provide the best performance.

Advantages of Service Programs

The advantages of service programs are:

- They do not take up auxiliary storage space. There is only one copy for all users.
- There is only a single copy of the read-only code in main storage for all users. In this respect, a service program is the same as a program that you call dynamically.

  Each user of the service program has an independent work area (just as in traditional OPM programs).

- Service programs can be maintained independently of the programs that use the functions. In most cases, changing a service program does not cause a program using the function to be changed or re-created.
- You can pass parameters to a service program by using the traditional parameter list or by importing and exporting variables.
- There is additional overhead the first time you call a program that refers to a service program that is not already activated. However, subsequent usage is just as fast as if you had tightly bound the function to the calling program. Thus, for a function like date conversion, a service program could fit very well if you have a high level of use. You have the code in one place, so if an update is needed, you can easily change that one place. A common function that multiple applications can use is an excellent candidate for a service program.

Possible Disadvantages of Service Programs

Service programs have disadvantages as well. Depending on how the service programs were designed, they could have some of these disadvantages:

- Depending on how you create them, there may be a little more work involved in creating service programs. The challenge lies in designing them to do what you want to do. It may take a little more design work to do the best job with a service program.
- Service programs are less desirable for a function you may or may not need. The reason is that it is slower to call a main program that refers to a service program.

  For example, you may have an exception-handling function that does not run if the program completes normally. If so, it is not a good performance choice to place the exception routine in a service program. If you have a routine you need only once (or a few times) in your application, it is also not a good performance choice for a service program.

Different Alternatives for Repetitive Functions

Designing different alternatives for repetitive functions in an ILE application is similar to what you do in a traditional OPM design. For example, if you have a repetitive function, you could do one of the following things with an OPM design.

- Code the same set of code multiple times in the same program
- Use a subroutine
- Use an include (like /COPY in RPG)
- Call a separate program

All of these approaches have design, testing, performance, and maintenance tradeoffs. No single solution is best for all situations. The same is true for ILE design approaches, except that you have more tools to use in your design.

It should be clear that there are no strict answers about what is the best design. Two good programmers could come up with different approaches and both could work well. Unless you are dealing with a very high degree of usage, most designs could be done effectively with several solutions.

Designing Service Programs

There is overhead in binding a service program when you activate the program that declares its use. (This is true when it is not the first time you call a procedure in the service program.) Thus, you generally want to have a service program that is capable of performing multiple functions. For example, the simple date conversion that converts from MMDDYY format to YYMMDD format would generally be too small for a single service program. A better approach is to have one service program that is capable of many different functions.

The overhead to bind a service program has both a constant cost and a variable cost that depends on how many symbols need to be resolved. Examples of symbols are:

Name of a variable to pass
Name of a procedure to call within the service program

If you pass information using a parameter list and do not export or import variables, the symbol list can be a list of your modules. However, it might not contain the names of all your modules.

One of the major considerations that you must make in the design of a service program involves your naming convention. In a traditional AS/400 design, program names only have to be unique in a library and you can use the same name again within a program. With the import and export of variables, you must be aware of all of the symbols that you are going to export.

This need to have unique export variable names does not occur if you pass parameter lists. When you use parameter lists, the only thing you export is the module names.

The term export indicates that which you are going to make known to the outside world from the service program. For example, if the sample date-conversion routine becomes a service program, you have to export the name of the program and the names of the input date (MMDDYY) and the output date (YYMMDD). You do not have to export the names of work fields in your program (for example, WORK4).

How Much Should You Package in One Service Program?

Generally speaking, you package utility functions. Utility functions tend to come in two forms:

Cross-application
A good example of a cross-application utility function is the date-conversion function. It can be used in a wide variety of applications that have no real connection. However, there may be simpler solutions to date conversion in RPG IV. (You can use new RPG operations, instead of calling a procedure.)

Application-specific
A good example of an application-specific utility function is a validity-checking function, relative to a certain field. For example, if you have multiple warehouses from which you can distribute, each warehouse in your order entry application probably has a unique ID. If the end user enters a warehouse ID, you probably have a standard routine that ensures the warehouse ID is valid.

You may have multiple programs associated with the order entry and inventory applications that need the same validity-checking routine for warehouse ID.

A reasonable design could include the complete range of:

- None, one, or more cross-application service programs
- None, one, or more application-specific service programs for each major application area.

The answer varies, depending on your application needs. You could well decide that service programs are not for you or not needed in a specific application area. For example, even if you have a warehouse ID validity-checking routine, you may prefer to bind the module into the using programs for performance reasons. Service programs are not always the answer.

Naming Convention for a Service Program

As with CRTPGM, there is no source member for a service program (except for binder language source), so you have to define a name to use. It is valid to use any of the module names as the name of the service program. Both objects could be in the same library because they are different object types.

Re-creating a Service Program

As when you create a program, you have to consider what strategy you are going to use if you have to create a service program again. Respecifying all the parameters is tedious and very error prone. In most cases you need to make only an internal change to one of the modules in a service program. If you are not changing the interface, you do not need to ask any where-used questions. For most situations, you just re-create the program without needing to consider what programs use the function.

However, if you add a procedure to a service program or add a variable being exported, you have to consider how to handle the change. For information about those considerations, see the *ILE Concepts* manual.

Replacing a Service Program Module

Replacing a service program module is very much like replacing a CRTPGM program module. You can do one of the following:

- Replace the entire program by using the same CRTSRVPGM command
- Replace one or more modules by using the UPDSRVPGM command

Assume that you had changed the MDYYMD module and wanted to replace it in the DATSRV1 service program. You would specify:

```
UPDSRVPGM  SRVPGM(DATSRV1) MODULE(MDYYMD)
```

The command defaults to use the same set of exports that you defined when the service program was originally created. If you are following these steps on the system, try the UPDSRVPGM command now.

You do not have to do anything with the programs that use functions in the service program. When the main-line program is activated, it binds the latest version of the

service program. This is a significant advantage of using service programs, instead of using modules bound directly into the program.

Which Activation Group Is Used

In our examples, the value specified on the ACTGRP parameter on the CRTPGM command was *CALLER. The default of *NEW was not used because normally you do not want a new activation group. In fact, you cannot use importing and exporting variables unless you are in the same activation group.

The default value for the activation group (ACTGRP) parameter on the Create Service Program (CRTSRVPGM) command differs from that on the Create Program (CRTPGM) command. The default value on the CRTPGM command is *NEW. The default value on the CRTSRVPGM command is *CALLER.

Because CRTSRVPGM defaults to ACTGRP(*CALLER), the service program runs in the activation group of the calling program. Using *CALLER allows the function to work but may cause performance problems on the call. It may be more desirable to have a specific activation group.

Using a CL Control Procedure in a Service Program

Many user applications begin with a CL program that controls the application and allows the full use of CL commands. You can use the same approach for a service program.

Assume you are using a parameter list approach for all usage. The picture would be like Figure 14-3 on page 14-7.

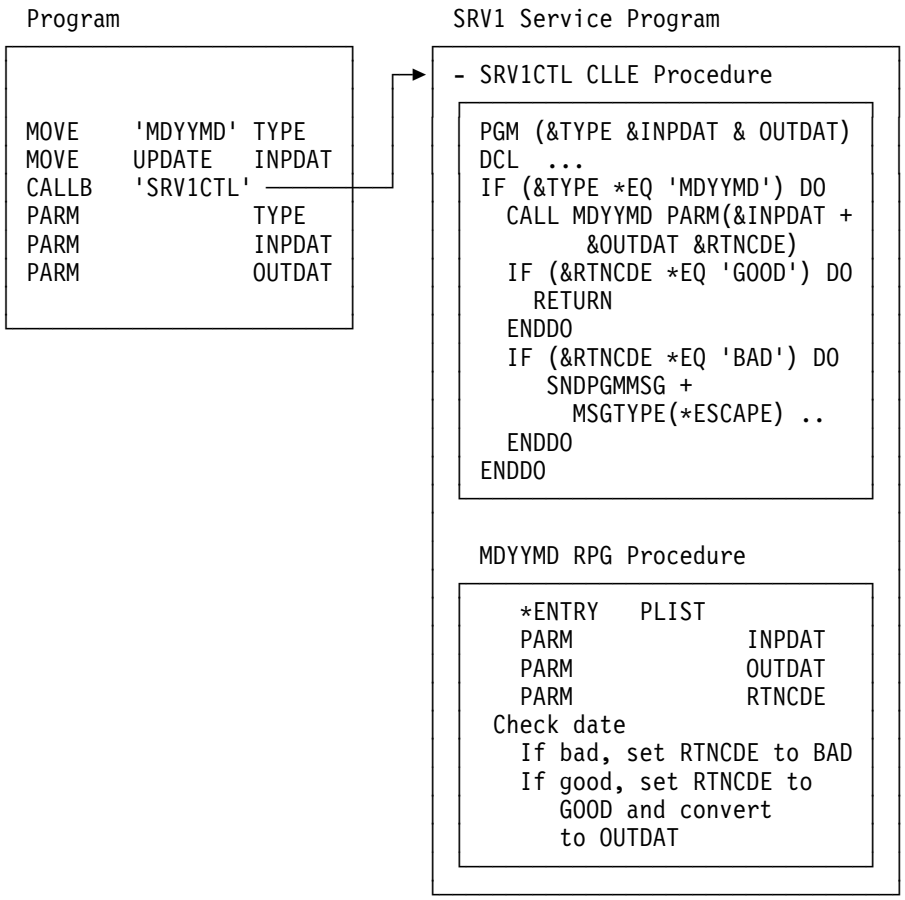

Figure 14-3. A Sample Service Program Using a CL Control Program

The advantages of this approach are:

- The CL procedure can perform any command. When the escape message is sent, it is sent directly to the main-line program. There is no guesswork about what will happen.
- The RPG interfaces (the procedures doing the real work) are hidden from the main-line procedures. To prevent the use of a CALLB to one of the RPG procedures, you have to use binder language.

The disadvantages are:

- The parameter list solution must be used to call the procedure in the service program. This is because it uses a CL program (CL does not have import and export syntax). Therefore, the full list of parameters on which the procedure in the service program can work must be passed every time. If you add a procedure that requires a new parameter, you will affect every program that is using the service program.
- An additional call occurs for every use of a procedure in the service program. Although it is a bound call from within the service program, it is still more overhead.

Which Program or Procedure Gets the Escape Message?

In an OPM call stack approach, an escape message is sent to the previous program. This is the program that called the function. Using ILE, the escape message gets special treatment. Assume that PGM1 calls MOD1. MOD1 can be any of the following:

- Another program (dynamic call if CALL was used)
- A procedure in the same program
- A procedure in a service program

Figure 14-4 shows the normal call stack approach using an escape message.

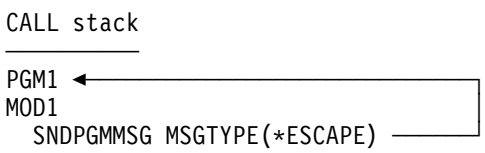

Figure 14-4. Normal Call Stack Approach Showing Escape Message

When the escape message is sent, MOD1 automatically ends (the message handler functions ends the program or procedure that sends the escape message). It is up to PGM1 to monitor for the condition or the message handler function ends that program. This concept is the same as it is with OPM programs.

When you call a service program, you might have many additional procedures that are called and are on the call stack when an escape message is sent (see Figure 14-5).

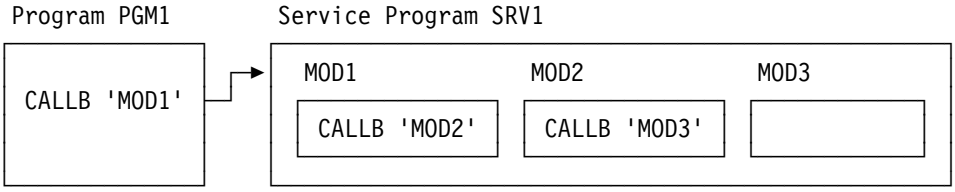

Figure 14-5. Calling a Service Program. Module MOD3 discovers an error and sends an escape message to the previous program.

The previous program should not be MOD2. It is just a middleman type of procedure in the service program. The system has code that tells it where the real program boundary is, so the system actually sends the escape message to PGM1. You can read more about this type of escape message handling in the *CL Programmer's Guide.*

Signature Checking

The system provides a function like level checking of database files to assist you in determining if there have been any changes between the interface of the calling program and the service program. Like database level checking, the technique used is to develop a hash number value (a signature) based on the exports from the service program. The value is placed in the object of the calling program when it is created and compared when the service program is activated. If you have defined your own signature, this type of signature checking is not necessary.

As is the case with database files, you can specify LVLCHK(*NO), in which case there is no signature checking. However, unlike the situation with database files, you can have multiple valid signatures for the same program. If you are supplying upward-compatible function, you can add a new signature but retain previous versions.

There are two ways to define what is exported from the service program:

- Use EXPORT(*ALL) on CRTSRVPGM. When you use EXPORT(*ALL), you are requesting two functions:
  - Export all procedure names and all variables identified for export.
  - Use signature checking.
- Use the binder language. The binder language is a series of simple source statements that lets you specify:
  - The symbols to be exported (both procedure and variable names)
  - Whether you want signature checking (a LVLCHK of *YES or *NO)
  - Multiple signatures

  You refer to the member that contains the binder language by specifying the following parameters on CRTSRVPGM.

```
EXPORT(*SRCMBR)
SRCFILE(lib/file)
SRCMBR(member)
```

Considerations for EXPORT(*ALL)

The service programs in the sample application were created by using EXPORT(*ALL) on CRTSRVPGM. This is a simple solution when showing examples or when testing.

The major disadvantage of EXPORT(*ALL) occurs when:

- You add (or remove) a procedure in the service program.
- You add (or remove) a variable that is identified for export.

Any change of this type causes a change to the signature of the service program. Because EXPORT(*ALL) uses the default of LVLCHK(*YES), every program that was using the service program has to be re-created. Re-creating a using program is the only way to get the new signature into the program. Note that the re-creation must occur even if the using program is not affected by whatever change occurred.

Because of this, the use of EXPORT(*ALL) is probably not a good choice for a typical service program. You need binder language.

Binder Language

You can use binder language instead of using the value EXPORT(*ALL) as was used in the sample application. EXPORT(*ALL) exports the procedure names and any variable names in which an RPG module used a D specification to define an EXPORT.

The default for the CRTSRVPGM EXPORT parameter is *SRCFILE. This means you use source statements to describe what should be exported outside of the

service program. You describe the source file and member to be used on the CRTSRVPGM SRCFILE and SRCMBR parameters.

The intent of the binder language is:

- To allow you to have procedures and exported variables within the service program that cannot be accessed directly. It allows you to describe the public interface to the program.
- To allow you to control upward-compatible changes being made to the service program. This is so there is no effect on existing programs that use the service program.

You use the binder language to describe the exports you will allow. The binder language is entered into a normal source member. A command-like syntax is used, but the commands are unique to the binder language and can be read only by the binder. They cannot be run from command entry. This concept is similar to the commands provided for command definition source.

The simplest binder language specifies a service program that supplied only a single procedure name as the public interface. It does not export any variables outside the service program. This is the case if you have a single function that communicates with other programs through the use of:

Parameter lists
Local data area
Database files
Data queues

If your only public interface is MODX, the binder language is:

```
STRPGMEXP
EXPORT SYMBOL(MODX)
ENDPGMEXP
```

This binder language specifies that the only external interface to the service program is the procedure MODX. Based on what the external user of the service program sees, that user would not know whether the service program contained one procedure or many procedures.

A more likely scenario is one in which you export variables between procedures within the service program. Therefore, you define the RPG modules with EXPORT and IMPORT definitions, but you do not want all of the variables made available to the public. For example, assume you want your service program to look like Figure 14-6.

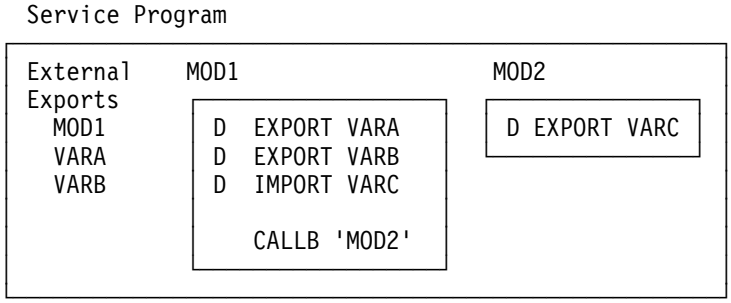

Figure 14-6. Service Program with Named Variables Passed between Modules

You do not want MOD2 or VARC to be known to anyone who is external to the service program. If you use EXPORT(*ALL) when you create the service program, anyone outside the program could do a CALLB to MOD2.

MOD1 does not use a parameter list for VARC for the CALLB to MOD2. Instead, the import and export of variables is used in each of the programs.

This is what the binder source for the simple example looks like:

```
STRPGMEXP PGMLVL(*CURRENT)
EXPORT SYMBOL(MOD1)
EXPORT SYMBOL(VARA)
EXPORT SYMBOL(VARB)
ENDPGMEXP
```

The symbols between a STRPGMEXP and an ENDPGMEXP binder language command define the public (external) interface to the service program and prevent a user from accessing MOD2 or VARC. You can have multiple STRPGMEXP commands that follow the PGMLVL(*CURRENT) value for previous versions. The intent of this support and of the LVLCHK parameter on the same command is to help software developers make upward-compatible enhancements on subsequent releases.

By defining PGMLVL(*PRV), you can provide a different signature that is still recognized as valid. In addition, you can define your own signature.

If you pass parameter lists to procedures in service programs and do not import and export variables, you probably still need binder language. It is needed to avoid the problem of adding procedures to (or removing procedures from) an existing service program.

QUSRTOOL Tool to Generate Binder Language

Rather than keying all of the binder source, you can use a tool in QUSRTOOL to generate the source and then tailor it with SEU. This tool extracts the exports from a module (or a set of modules) and places the result into a source file. It is in a format compatible with the binder language. (See member TBNINFO in file QATTINFO in QUSRTOOL for information about how to create and use this tool.)

Binding Directory

A binding directory is optional. Binding directories are essentially lists of module and service program objects that you may need when creating ILE programs or service programs. These names of *MODULE and *SRVPGM objects may be needed for CRTPGM or CRTSRVPGM to run successfully. *PGM names cannot be specified in a binding directory.

Service programs or modules listed in binding directories are used only if they provide an export that can satisfy any currently unresolved import requests. A binding directory is a system object and is identified to the system by the symbol *BNDDIR.

Binding directories are used for reasons of convenience and program size. With binding directories, you do not need to name a list of modules every time you create a program or service program. For example, assume you know you want a

specific procedure. You do not have to know what service program the procedure is in as long as the service program is described in a binding directory. Also, there are very few restrictions placed on the entries in a binding directory.

You describe which binding directory to use on the BNDDIR parameter of CRTPGM. What you do have to know is the name of the procedure to call and the list of variables to import, or the parameters to pass. You also have to define the attributes of the variables in your program. The BNDDIR parameter is supported on both CRTPGM and CRTSRVPGM.

Importing and Exporting Variables

In this example, you eliminate the parameter list and use EXPORT and IMPORT statements to describe the variables. There are a few restrictions to importing and exporting variables:

- CL does not support the technique. CL supports only parameter list passing. This does not prevent you from including a CL module in a service program or from calling a service program from CL. It just means that you must use a parameter list.

  The system does not restrict a service program from using either a parameter list or from importing and exporting variables. It can use both. In fact, the same procedure can be called with both techniques.

- A program cannot export a variable to a service program. There is a one-way street between a program and a service program (see Figure 14-7).

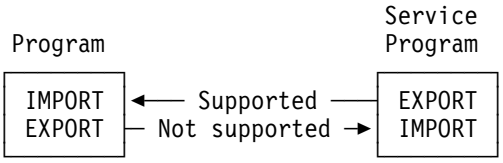

Figure 14-7. Communication between a Program and a Service Program

There is a simple way around this restriction. It is to have the calling program move a value to the variable for importing and then to call the procedure in the service program. This coding is similar to that done for many applications that pass parameter lists. For example:

```
CALLB  'SRVPGM'
PARM               MMDDYY
```

The term export refers to the program that has the storage for the data. The thing to keep in mind is that a program can import only from a service program (see Figure 14-8).

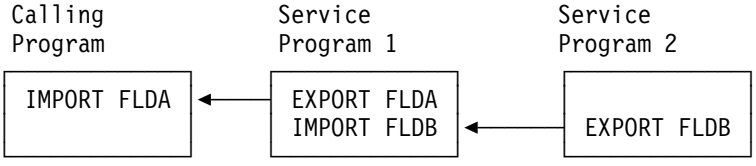

Figure 14-8. A Calling Program and Two Service Programs

You can string together many service programs, but you must be aware of the performance implications of activation. In RPG, you specify IMPORT and EXPORT in the new D specifications. There are a few differences from parameter list passing if you define a field as IMPORT:

- The field cannot be initialized with the RESET operation.
- The field cannot be a compile-time array.

After a program is started, export fields are under complete control of the programmer. No action to end the program (except ending the activation group) reinitializes the export field. Returning with LR on or ending because of an exception and then calling the program do not affect the export fields.

A service program can export only to a program that is in the same activation group. If the service program is in a different activation group, you can still use parameter lists.

You must ensure that any programs that use the service program are in the same activation group. When you use CRTSRVPGM, the OPTION parameter defaults to require unique procedure names and unique export variable names. You must take this option if you are using RPG. Otherwise, you could not access the duplicates. Because of this requirement, you need a good naming convention to avoid potential problems.

The sample service program that we already created contains two date-conversion modules. Both modules need the following variables:

MMDDYY
YYMMDD
SETLR

Because of the CRTSRVPGM default, the module names have to change. The requirement exists regardless of what you use from the service program. If you do not change the names, you cannot create the service program. Table 14-1 shows the module names used for the new example:

Table 14-1. Names of Service Program Modules Changed Due to RPG Restrictions. The variables in the service program modules need different names under each function.

Old Names	**MDYYMD Function**	**YMDMDY Function**
MMDDYY	CV1INP	CV2OUT
YYMMDD	CV1OUT	CV2INP
SETLR	CV1LR	CV2LR

The new example looks like Figure 14-9 on page 14-14.

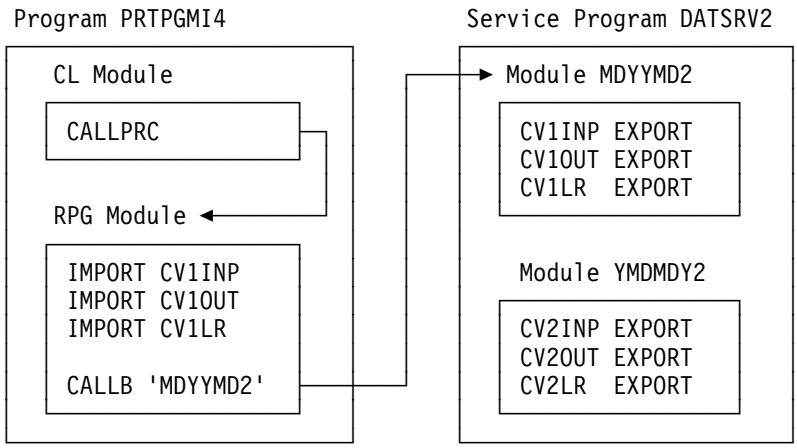

Figure 14-9. Date-Conversion Service Program

Using the Import and Export of Variables

CL is not calling a procedure in the service program directly. Instead, the CL program uses a CALLPRC command to call the main-line RPG procedure. If the CL module does need a procedure in the service program, it must use a parameter list. This is true because CL does not support the import and export of variables.

The calling program source does not name the service program. It names a procedure. This provides good flexibility in that you can change your mind and make the MDYYMD2 procedure a part of some other service program. You can also make it part of the main-line program rather than part of the service program. The code in the program (CALLB and the importing of variables) remains the same.

Creating the Service Program

If you are coding these examples on a system as you read through this book, follow these steps to create a service program.

1. Use PDM to add a new member named MDYYMD2 with a type of RPGLE and a text description of `Date conversion module - pass vars - MDY format to YMD`. When the SEU Edit display appears, enter the following:

```
 * MDYYMD2 - Date conversion - export - MDY format to YMD
 *
D CV1INP          S              6    EXPORT
D CV1OUT          S              6    EXPORT
D CV1LR           S              4    EXPORT
C                   MOVE      CV1INP        WORK2             2          Move YY
C                   MOVEL     WORK2         CV1OUT                       Move YY
C                   MOVEL     CV1INP        WORK4             4          Move MMDD
C                   MOVE      WORK4         CV1OUT                       Move MMDD
C     CV1LR         IFEQ      '*YES'                                     If *YES
C                   SETON                                          LR      Set LR
C                   ENDIF                                                  If *YES
C                   RETURN                                                 Return
```

2. End SEU and use PDM option 15 to create the RPG module.
3. Use PDM to add a new member named YMDMDY2 with a type of RPGLE and a text description of `Date conversion module - pass vars - YMD format to MDY`. When the SEU Edit display appears, enter the following:

```
 * YMDMDY2 - Date conversion - export - YMD format to MDY
 *
D CV2INP          S              6    EXPORT
D CV2OUT          S              6    EXPORT
D CV2LR           S              2    EXPORT
C                   MOVE      CV2INP        WORK4             4          Move MMDD
C                   MOVEL     WORK4         CV2OUT                       Move MMDD
C                   MOVEL     CV2INP        WORK2             2          Move YY
C                   MOVE      WORK2         CV2OUT                       Move YY
C     CV2LR         IFEQ      '*YES'                                     If *YES
C                   SETON                                          LR    Set LR
C                   ENDIF                                              If *YES
C                   RETURN                                             Return
```

4. End SEU and use PDM option 15 to create the RPG module.
5. Create the service program by using the following CRTSRVPGM command:

   ```
   CRTSRVPGM   SRVPGM(DATSRV2) MODULE(MDYYMD2 YMDMDY2) EXPORT(*ALL)
               TEXT ('Uses import and export of variables')
   ```

   Because EXPORT(*ALL) is specified, the system finds all the EXPORT statements and builds a list of them along with the procedure names. This export list contains the only variables that can be imported from the service program.

Creating the Calling Program

To create a calling program, follow these steps:

1. Use PDM to add a new member named PRTPGMC8. Make it type CLLE and use a text description of `Print program - uses service pgm - export`.
2. When the SEU edit display appears, use function key 15=Browse/Copy to copy the source from the PRTPGMC7 source.
3. Change the first comment to:

   ```
   /* PRTPGMC8 - Print program - uses service pgm - export */
   ```

4. Change the CALL command to:

   ```
   CALLPRC   PRC(PRTPGMR8) PARM(&LIB)
   ```

   This is the call to the main-line RPG procedure and not to a procedure in the service program. It still uses a parameter list technique.
5. End SEU and use PDM option 15 to create the module.
6. Use PDM to add a new member named PRTPGMR8. Make it type RPGLE and use a text description of `Print program - uses service pgm - export`.
7. When the SEU edit display appears, use function key 15=Browse/Copy to copy the source from the PRTPGMR7 source.
8. Change the first comment to:

   ```
   * PRTPGMR8 - Print program - uses service pgm - export
   ```

9. Change the CALL operation to:

   ```
   CALLB  'MDYYMD2'
   ```

10. Delete the PARM statements because import and export of variables is used instead.
11. Add the following D specs before any C specs.

```
D CV1INP     S             6 IMPORT
D CV1OUT     S             6 IMPORT
D CV1LR      S             4 IMPORT
```

12. End SEU and use PDM option 15 to create the module.

    The source does not name the service program. It names only a procedure.

13. Use the CRTPGM command to create the program object PRTPGMI4.

```
CRTPGM   PGM(PRTPGMI4) MODULE(PRTPGMC8 PRTPGMR8)
         ENTMOD(PRTPGMC8) BNDSRVPGM(DATSRV2) ACTGRP(*CALLER)
         TEXT('Print program - uses serv pgm - export')
```

14. Call the program:

```
CALL    PGM(PRTPGMI4) PARM(PRTLIB)
```

15. Display the spooled file that was produced. You see an additional module, program, and service program object.

Appendix A. Packaging CRTPGM and CRTSRVPGM

You could respecify the correct Create Program (CRTPGM) or Create Service Program (CRTSRVPGM) parameters each time you need one of the commands to re-create a program. However, you should consider a solution that allows you to specify the parameters once and then to reuse that same solution. PDM does not have an option for this kind of function. One solution is Application Development Manager/400.

The system does not directly support a solution, but there are a few techniques you might use to do so. Before looking at these techniques, it is important to understand how you affect the library list when you create objects and the importance of the tailoring module and the program objects.

Using the Library List

When a command like CRTPGM or CRTSRVPGM is used to create modules, the MODULE parameter is a list of qualified module names. The library part of each qualified name defaults to *LIBL (the library list). Consequently, you can use the library list to assist you by incorporating a test version of a module.

For example, assume that you have library PRODUCTION, which contains both module and program objects (see Figure A-1).

```
Library PRODUCTION
------------------
-  MOD1  *MODULE
-  MOD2  *MODULE
-  PGM1  *PGM
     Contains
       MOD1
       MOD2
```

Figure A-1. Library PRODUCTION with Both Module and Program Objects

When you create PGM1, the CRTPGM command could look as follows:

```
CRTPGM  PGM(*CRTDFT/PGM1) MODULE(MOD1 MOD2) ...
```

If you do not specify a library for the module objects, the system finds them by using the library list. For example, suppose library PRODUCTION was specified as the current library (for example, on the CURLIB parameter for the Change Library List (CHGLIBL) command). Then, when you run the CRTPGM command, the system uses modules MOD1 and MOD2 from library PRODUCTION and creates the program object in library PRODUCTION.

Assume that you want to test a new version of MOD2. You create the new version in library TEST. At that point, the TEST library appears as it does in Figure A-2.

```
Library TEST
------------
-  MOD2  *MODULE
```

Figure A-2. Library TEST with Modules

If you change CURLIB to library TEST, but the library list also includes the library PRODUCTION, the system uses the search order shown in Figure A-3 on page A-2.

```
Current library             TEST
Other libraries on LIBL      .
                            PRODUCTION
                             .
```

Figure A-3. Search Order When Library List Includes Library PRODUCTION

Then you create a different version of PGM1 in library TEST, using the same CRTPGM command that was shown earlier. Because of the library list search order, the system uses module MOD1 from library PRODUCTION and module MOD2 from library TEST. When packaging the CRTPGM and CRTSRVPGM commands, consider a solution that allows you to create both production and test versions of both modules and programs.

Tailoring Modules and Programs

Because you may be combining the same module into several different programs, you want the module to be as small as possible. After you test a module that is used in many different programs, you may want to minimize the size of the object and optimize the code before you bind it into several programs.

The minimization and optimization functions exist with OPM. However, the benefits are even greater if you use them on ILE modules and programs.

Consequently, when you create a production module to be used in several programs, you may want to use the Change Module (CHGMOD) command to tailor the module. For example, you can remove the observability of a module to minimize the size even though you will be unable to debug the module. For well-tested functions that appear in multiple programs, this is probably a good tradeoff. You may also want to use the OPTIMIZE parameter on CHGMOD to ensure the best performance.

If you are using CRTPGM with several modules, some modules may have observability and some may not. You may want to keep the observability for certain modules in a production environment in order to allow a formatted dump if an error occurs.

You can minimize the space of the observability by using the RMVOBS parameter on CHGPGM to remove the observability of all modules in a program. You can also use the Compress Object (CPROBJ) command to compress observability. You can also keep the module and program templates (the *CRTDTA part of observability). Depending on whether you use the CHGMOD or CHGPGM command, you can remove the *DBGDTA part of observability. This allows you to retranslate the object, if needed.

You may currently create production objects by using a simple replace function such as option 14 from the PDM menu. With ILE, there is a greater benefit if you use a more tailored solution. You can run a unique set of commands for each of your production program and module objects.

Input Stream

When you use CRTPGM, there is no source member for the program that you create from a series of modules. Using CRTPGM, name the modules you want to include, and identify one of them as the program entry point (PEP).

You could make a source member of type CL using the member name as the name of the program. Then you could make a job stream of the command or commands that you want to run, as in Figure A-4.

```
//BCHJOB      JOB(xxxxxx) JOBD(xxxx)
CRTPGM         ....
//ENDBCHJOB
```

Figure A-4. Job Stream Using a CL Type Source Member

A command like SBMDBJOB can be used to submit the commands to batch:

```
SBMDBJOB    FILE(xxx) MBR(yyy)
```

You can create your own PDM option for the SBMDBJOB command. Use CRTSRVPGM, as the service program is usually the program that you want to create. If you want to use a job stream to package the CRTSRVPGM command, you probably want a separate source file and the same member name.

Creating a CL Program by Using CRTPGM or CRTSRVPGM

You can make a CL program with the proper CRTPGM or CRTSRVPGM commands and call that program when you want to create the program. For example, you can enter the source as:

```
PGM
CRTPGM     PGM(CRTPGMI2) ....
```

Create this as a normal CL program (either OPM or ILE). Create your own PDM option to submit a job to call the program.

A significant advantage for this approach is that you can use the full power of CL programs. For example, you determine whether the modules exist, and then create them if they do not. The code might look like this:

```
PGM
CHKOBJ     OBJ(PRTPGMC5) OBJTYPE(*MOD)
MONMSG     MSGID(CPF9801) EXEC(DO) /* Not found */
CRTCLMOD   MOD(PRTPGMC5) ...
ENDDO      /* Not found */
  .
  .
CRTPGM     PGM(CRTPGMI2) ....
```

A difficulty of this approach is how you name the member and the program. You cannot use the same program name because you already have a program by that name. Either you have to use a different name, or you have to use the same name in a different library.

You could have a separate library of source and objects in which you perform all the CRTPGM/CRTSRVPGM work. However, a disadvantage of this solution is that there are significantly more source members and program objects.

Standard Create Program for All Uses of CRTPGM and CRTSRVPGM

Instead of creating a separate member and separate program for each CRTPGM/CRTSRVPGM you need, you can have a single CL source member and program do the work for multiple objects. You can pass it the name of the program or module you want to create. For example, a program named CRTPGMC looks like the following:

```
/* CRTPGMC - Create steps for ILE objects                    */
/*                                                           */
             PGM        PARM(&OBJECT)
             DCL        &OBJECT *CHAR LEN(10)
             IF         (&OBJECT *EQ 'PRTPGMI2') GOTO PRTPGMI2
             IF         (&OBJECT *EQ 'XXXX    ') GOTO XXXX
             SNDPGMMSG  MSGID(CPF9898) MSGF(QCPFMSG) MSGTYPE(*ESCAPE) +
                          MSGDTA('No such object name as ' +
                          *CAT &OBJNAM *TCAT ' exists to create. +
                          Correct the name or change the CRTPGMC +
                          source to include the name')
 PRTPGMI2:               /* Prints list of program and module types */
               .
               .
             CRTPGM     PGM(CRTPGMI2) ....
             GOTO       ENDPGM
 XXXX:                   /* The XXXX program */
               .
               .
             CRTSRVPGM  PGM(XXXX) ....
             GOTO       ENDPGM
               .
               .
 ENDPGM:     ENDPGM
```

You just add another IF statement and label routine for each use of CRTPGM/CRTSRVPGM/CRTxxxMOD.

You can use the full benefit of CL programming to determine if modules exist. Then, if they do not, you can create them before you get to the CRTPGM or CRTSRVPGM commands. For example, the code for PRTPGMI2 might look like the following example. After the create command, you can use the CHGMOD command.

```
PRTPGMI2:               /* Prints list of program and module types */
            CHKOBJ      OBJ(PRTPGMC5) OBJTYPE(*MOD)
            MONMSG      MSGID(CPF9801) EXEC(DO) /* Not found */
            CRTCLMOD    MOD(PRTPGMC5) ...
            CHGMOD      MOD(PRTPGMC5) OPTIMIZE(*FULL) RMVOBS(*ALL)
            ENDDO       /* Not found */
              .
              .
            CRTPGM      PGM(CRTPGMI2) ....
```

The CRTPGMC program should run in batch. You can write a small CRTCRTPGMC program to submit a batch job for this function.

```
/* CRTCRTPGMC  - Submit a job to use CRTPGM  */
             PGM        PARM(&OBJNAM)
             DCL        &OBJNAM *CHAR LEN(10)
             DCL        &MSG *CHAR LEN(80)
             SBMJOB     JOB(&OBJNAM) +
                          CMD(CALL PGM(CRTPGMC) PARM(&OBJNAM))
             RCVMSG     MSGTYPE(*LAST) MSG(&MSG)
             SNDPGMMSG  MSG(&MSG) MSGTYPE(*COMP)
             ENDPGM
```

You can also write a command interface for this with the following command definition source:

```
/* CRTCRTPGM - Create CRTPGM command                          */
/*    CPP is CRTCRTPGMC                                       */
/*                                                            */
             CMD        PROMPT('Create CRTPGM - USR')
             PARM       KWD(OBJ) TYPE(*NAME) LEN(10) +
                          EXPR(*YES) MIN(1) +
                          PROMPT('Object name')
```

Create the command as follows:

```
CRTCMD    CMD(CRTCRTPGM) PGM(CRTCRTPGMC)
```

When you want to create or re-create the PRTPGMI2 program, you use the following command:

```
CRTCRTPGM  OBJ(PRTPGMI2)
```

This command submits a batch job to re-create the program based on the code you placed in the PRTPGMC CL program. For many application uses, you may want to use one set of create commands for production objects and another set for test objects. For example, you may want to create the same program using test source or place the object in a test library. The CRTCMD is not appropriate for this approach. Instead, you can use the library list, as described in “Using the Library List” on page A-1.

Another approach is to make variable names of the source files and object libraries used throughout the source for CRTPGMC and to pass the names in from the command.

In addition, you may want separate CRTPGMC programs for different application areas. You can add a parameter to the sample command to describe which library the program was in, or you can use a different program name.

Another alternative is the MAKE process. Setting it up takes some effort, but once set up, it has the potential to be superior to the methods mentioned above. In the era of modular programming, handles these problems without having to re-create the program if the source is updated.

Appendix B. Where-Used Capability

In most programming environments, there is a need to ask typical questions such as:

Which programs use FILEA?
Which programs update the file FILEA?
Which programs specify the field SALARY?
Which programs use the CPYF command?
Which programs use the subprogram PGM1?

These questions are typical of a what are normally referred to as where-used questions. The need to answer where-used questions exists in any system and is generally greater with a system like the AS/400, even with a traditional OPM design.

As you start using more modularity, the need for a where-used function becomes greater. ILE introduces some new where-used questions. This chapter discusses techniques for you to use in answering both the old questions and the new questions.

Print Command Usage

The system supports the Print Command Usage (PRTCMDUSG) command, which provides a spooled listing of the programs that use a specified command. For example, if you want a list of the programs in PRTLIB that use CALL, you specify:

```
PRTCMDUSG   CMD(CALL)  PGM(PRTLIB/*ALL)
```

PRTCMDUSG is a simple function, but it works only for commands. It does not allow you to create a subset of the list, such as to ask which programs call DATCVT. PRTCMDUSG is helpful but fairly limited.

Scanning Source

If you have good control of your source (you know it represents your production system), a simple scanning of your source may be an adequate solution. For example, if you type the examples described in this book and want to know what programs call DATCVT, you can just scan the source.

The standard way to scan source is the Find String PDM (FNDSTRPDM) command. This command is part of PDM and supports many options. If you want to see the member names and the source statements where you used DATCVT, specify:

```
FNDSTRPDM    STRING(DATCVT) FILE(PRTLIB/SOURCE)
             OPTION(*NONE) PRTRCDS(*ALL)
```

You can try the command and see the results yourself. You get a listing that describes each of the members in which DATCVT was found, including member-level information, the specific statements that have DATCVT in them, and a summary of the number of records in the member and the number that contain DATCVT.

Any time you scan source, you are going to find not only the statements you are interested in, but other statements as well.

Scanning source tends to give:

- Extraneous noise output, such as comments within your source that refer to the string
- Same string value used to mean something else (for example, it could have been used as a field name in some program). Unless you are dealing with a lot of source, a scan can generally produce a good answer without significant difficulty.

If you have to change a module that is used in several programs, there is a simple solution. If you scan the source and re-create the steps, the result may be just what you need. If your environment is more complex, you should consider automatic searching and re-creation.

DSPPGM Command and API

The Display Program (DSPPGM) command displays the attributes of a program. These attributes include:

- Whether observability was removed
- The modules that were used

DSPPGM does not support an output file that you could use as a database file of the modules. The QBNLPGMI API provides the format PGML0100, which includes a user space of the list of module information. Using the API, it would be possible to develop a tool that generates a database file of modules used by a program or by all programs in a library.

DSPSRVPGM Command and API

The Display Service Program (DSPSRVPGM) command provides output about a service program and can also provide a list of the module objects that are bound to the program. There is no output file from the command.

The QBNRSPGM API supports the same information. Using the API, it would be possible to develop a tool that generates a database file of modules that are used by a service program or all service programs in a library.

DSPPGMREF Command

The Display Program Reference (DSPPGMREF) command provides a list of objects that are used by a program. An output file capability is available. You can also produce an output file of all programs in a library.

For example, you could specify:

```
DSPPGMREF   PGM(PRTLIB/*ALL) OUTPUT(*OUTFILE)
            OUTFILE(PRTLIB/PGMREFP)
```

One record is created for each reference to an object by a program. For example, the output for the PRTPGMR RPG program is two records. One describes the use of the QADSPOBJ file and the other is for the QPRINT file.

If you use overrides and variable names for references to objects, the information that is created becomes somewhat meaningless. The output does not represent what the application is actually using. An example is that the PRTPGMR program does not really use the QADSPOBJ file. Instead, an override directs the program to a file in QTEMP.

You can build access paths over the various fields and produce a reasonable where-used listing or allow queries to occur. This gives you an answer to the question of what programs use DATCVT. However, how complete the answer is depends on whether you always call the program with the constant DATCVT or use a variable name. The advantage of DSPPGMREF is that it is generated from your actual objects, which may be more accurate than looking at source.

Appendix C. Concept Review

The Integrated Language Environment (ILE) delivers some very significant advantages over the original program model (OPM). ILE offers capabilities for application writing that are simply not available in an OPM environment. As with most new programming functions, you want to do simple tasks until you gain experience. Here is a review of the important concepts covered in this book.

Concept Review

If some of these points are unclear, just go back and review the concepts.

- You are probably already using modular program design to some degree. ILE allows you to make additional use of modular design.
- Adopting a more modular design than the one you use with typical OPM applications will probably not improve performance.
- RPG IV has significant new function that can provide advantages to you, with or without using a more modular design. If you are an OPM RPG programmer, you should become familiar with RPG IV.
- CL, RPG, and COBOL use the following definition:

  ```
  One source member = One module = One procedure = One entry point
  ```

  This definition does not apply to ILE C/400.
- The source type you use with the programming development manager (PDM) identifies the type of syntax checking done in SEU and the type of CRT command you should use when you use a simple PDM option.
- With ILE, you create a module before you create a program.
- You cannot make a dynamic call to a module. Each language supports an easy-to-use command to compile a module and turn it into a program. Examples are CRTBNDRPG or CRTBNDCL. The modules created by these commands exist only temporarily.
- Each language supports a command, such as CRTRPGMOD or CRTCLMOD, that creates just a module. You bind modules together with the CRTPGM command.
- You can make various types of calls. Each is used in a different situation.
  - Dynamic calls can call only program objects.
  - Bound calls can call either:
    - A procedure in the same program
    - A procedure in a service program

  Each of these types of calls has both functional and performance implications. You should understand the advantages and the disadvantages, as well as the performance aspects.

  A service program concept can be a good middle ground, but it is not a good performance choice unless you are going to have at least one thousand uses in a given activation. Activating a service program at the beginning of the day or at a major application boundary may achieve good performance and provides the functional benefits of a service program.

Each high-level language supports a different type of statement to distinguish between a dynamic call and a bound call. For example, RPG uses CALL and CALLB, while CL uses CALL and CALLPRC. The source code does not distinguish between a bound call to a procedure in a module in the same program or to a module in a service program. This allows you to switch more easily from one technique to another.

- You cannot take the defaults from PDM to use CRTPGM or CRTSRVPGM, as you can when you use CRTRPGPGM or CRTCLPGM. Some parameters (MODULE, for example) must be entered (they cannot be defaulted). Use the defaults for some of the other parameters only when you fully understand them.
- Service program functions can be accessed by using the traditional parameter list or by importing and exporting variable names. CL supports only the traditional parameter list approach. However, you can use import and export for communicating to a service program only if it is in the same activation group.

Index

Special Characters

A

B

C

D

E

F

H

I

L

M

N

O

P

Q

R

Reader Comments—We'd Like to Hear from You!

AS/400
ILE Application Development Example
Version 4

Publication No. SC41-5602-00

Overall, how would you rate this manual?

	Very Satisfied	Satisfied	Dissatis-fied	Very Dissatis-fied
Overall satisfaction				

How satisfied are you that the information in this manual is:

Accurate				
Complete				
Easy to find				
Easy to understand				
Well organized				
Applicable to your tasks				
THANK YOU!				

Please tell us how we can improve this manual:

May we contact you to discuss your responses? __ Yes __ No
Phone: (____) _____ Fax: (____) _____ Internet: _____

To return this form:

- Mail it
- Fax it
  United States and Canada: **800+937-3430**
  Other countries: **(+1)+507+253-5192**
- Hand it to your IBM representative.

Note that IBM may use or distribute the responses to this form without obligation.

Name

Address

Company or Organization

Phone No.

Reader Comments—We'd Like to Hear from You!
SC41-5602-00

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

ATTN DEPT 542 IDCLERK
IBM CORPORATION
3605 HWY 52 N
ROCHESTER MN 55901-9986

Fold and Tape **Please do not staple** Fold and Tape

SC41-5602-00

Cut or Fold
Along Line

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC41-5602-00

Spine information:

AS/400 ILE Application Development Example *Version 4*